Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Češnjevar

Konvergenca relacijskih in nerelacijskih podatkovnih baz

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Matjaž Kukar

Ljubljana, 2015

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Jan Češnjevar, z vpisno številko **63090208**, sem avtor magistrskega dela z naslovom:

Konvergenca relacijskih in nerelacijskih podatkovnih baz

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom doc. dr. Matjaža Kukarja,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 18. marca 2015 Podpis avtorja:

Zahvaljujem se mentorju, doc. dr. Matjažu Kukarju, za strokovno pomoč in usmerjanje pri izdelavi magistrske naloge.

Kazalo

Seznam uporabljenih kratic

kratica	razlaga kratice
ACID	Atomarnost, konsistentnost, integriteta, trajnost
CPE	Centralna procesna enota
CQL	Cassandra Query Language
JSON	Java Script Object Notation
NoSQL	Not Only SQL
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
OQGraph	Open Query Graph
SQL	Structured Query Language
SUPB	Sistem za upravljanje s podatkovno bazo
YCSB	Yahoo Cloud Serving Benchmark

Povzetek

Cilj magistrskega dela je pregledati področje nerelacijskih in relacijskih podatkovnih baz in ugotoviti, ali trenutni trendi kažejo na zbliževanje teh dveh polov. Magistrsko delo vsebuje sistematičen pregled prenesenih konceptov iz relacijskih sistemov v nerelacijske in obratno, v katerem so preneseni koncepti ovrednoteni s primerjalnimi ocenami. Poleg tega so opisani sistemi NewSQL (VoltDB, FoundationDB) in podana ugotovitev, ali ti sistemi predstavljajo sintezo obeh svetov. V tem delu smo predstavili relacijske, nerelacijske in NewSQL sisteme, opisali smo njihove prednosti in slabosti. Iz dokumentacij in člankov smo odkrili prenesene koncepte, ki so se pojavili v letu 2014 na sistemih PostgreSQL, MariaDB, MySQL, FoundationDB in VoltDB, jih na sistematičen (podoben in kratek) način opisali in podprli s primeri programske kode. Vzpostavili smo virtualno okolje in razvili enostavne primerjalne teste v programskem jeziku Java za večino prenesenih konceptov. Kvalitativno smo ocenili koncepte z merjenjem časa izvajanja operacij in kvalitativno s prikazovanjem uporabe konceptov. Glavni prispevek magistrskega dela je sistematičen opis prenesenih in testiranih konceptov z zaključkom, kjer smo podali lastno mnenje, ali se sistemi med seboj res zbližujejo.

Ključne besede

Relacijske in nerelacijske podatkovne baze, NoSQL, NewSQL, konvergenca, FoundationDB, primerjalni testi

Abstract

The goal of this thesis is to review modern open-source relational and nonrelational databases and to discover if current trends are converging these two types of databases. The thesis contains systematic description of transferred concepts from relational to nonrelational databases and the other way around. Transferred concepts are also evaluated with benchmark tests. Thesis contains description and evaluation of NewSQL systems and provides opinion whether they contain best of both worlds. We describe relational, nonrelational and NewSQL systems, their weaknesses and strengths. From articles and documentations we extract concepts which lately appeared in PostgreSQL, MariaDB, MySQL, FoundationDB and VoltDB. We describe them in a systematic (similar and short) way and support our claims with examples of program code. We set up virtual environment and developed series of simple benchmark tests in Java for most of transferred concepts. Concepts are evaluated with quantitative tests (speed of operations) and qualitatively (examples of use). At the end there is our analysis of current state of convergence between systems. The main contribution of thesis is a systematic description of transferred concepts compiled in a single document, supported with test results.

Keywords

Relational databases, NoSQL, NewSQL, convergence, FoundationDB, benchmarks

Poglavje 1

Uvod

Upravljanje podatkov je ključno za vsako podjetje. Izbira pravilnega sistema za upravljanje s podatkovno bazo (SUPB) lahko ključno vpliva na kakovost poslovanja podjetja. Trenutno sta na področju sistemov za upravljanje s podatkovnimi bazami (SUPB) najbolj priljubljeni dve vrsti sistemov: relacijski in sodobni nerelacijski (NoSQL). Obe vrsti SUPB imata svoje značilnosti, prednosti in slabosti. Relacijski SUPB omogočajo shemo ACID, uporabljajo SQL in so zrela tehnologija. Nasprotno nerelacijski sistemi omogočajo hiter dostop, skalabilnost in delo z nestrukturianimi podatki. Ugotavljamo, da prednosti enih SUPB predstavljajo slabosti drugih SUPB in obratno. S prehodom iz relacijskega na nerelacijski SUPB ali obratno bi tako izgubili prednosti predhodnega, zato so začeli relacijski in nerelacijski SUPB nadgrajevati svoje sisteme z lastnostmi, značilnimi za drug tip sistemov. Trenutno se na tem področju veliko raziskuje. Eden izmed prenesenih konceptov je podpora nestrukturiranim podakom v relacijskih sistemih [1]. Nekateri avtorji [2] so raziskali področje relacijskih in NoSQL baz ter s preučevanjem prednosti in slabosti ugotovili, da je uporaba izključno enega sistema za vse probleme neprimerna.

Dandanes so podatki na voljo v poljubnih oblikah in obstaja množica sistemov, ki zna te podatke obvladovati. Termin "konvergenca" pomeni zmanjševanje razlik, ki delijo kaj enotnega, oziroma zbliževanje (povzeto iz SSKJ). V okviru naslova "konvergenca relacijskih in nerelacijskih podatkovnih baz" želimo narediti pregled področij relacijskih in nerelacijskih baz in ugotoviti, kako prepletena sta ta dva pola. Skušali bomo ugotoviti, ali bo zmanjšanje razlik med relacijskimi in nere-

lacijskimi podatkovnimi bazami privedlo do novega tipa podatkovnih baz, kako močna je konvergenca, kako sistemi dosegajo zbliževanje, in s primerjalnimi testi oceniti trenutno stanje na tem področju.

V magistrskem delu pregledamo koncepte, ki so se začeli prenašati iz relacijskih v nerelacijske sisteme in obratno. Opišemo jih na sistematičen način in jih ocenimo s primerjalnimi testi. Struktura magistrskega dela je sledeča.

V drugem poglavju opisujemo relacijske in nerelacijske sisteme, razložimo njihove prednosti in slabosti, opredelimo koncepte, ki so se začeli pojavljati v obeh sistemih, in opisujemo sisteme NewSQL kot sisteme, ki naj bi odpravili pomanjkljivosti relacijskih sistemov glede na zasnovo iz sedemdesetih let.

V tretjem poglavju na sistematičen način opisujemo prenesene koncepte iz relacijskih v nerelacijske sisteme in obratno.

V četrtem poglavju omenjamo problem standardizacije primerjalnih testov. Prenesene koncepte nato ocenjujemo kvalitativno (udobnost uporabe) in kvantitantivno (hitrost izvajanja operacij).

V petem poglavju podajamo sklepne ugotovitve napram hipotezam, postavljenim v prvem poglavju, in vsakemu konceptu dodamo lastno mnenje. Na koncu podajamo kritiko dela, možne izboljšave in omenjamo shrambe z več modeli, SQL++, Google F1 in standardne primerjalne teste kot področje, ki bi bilo zanimivo za nadaljnje raziskave.

Poglavje 2

Relacijski in nerelacijski sistemi za upravljanje s podatkovnimi bazami

Glavni namen dela je sistematično opisati prenesene koncepte in jih ovrednotiti s primerjalnimi testi. Postavili smo si sledeče hipoteze:

(a) Konvergenca se zaustavlja, saj se zdi, da se prenašajo le značilnosti, ki so zanimive; čaka se na naslednje zanimive koncepte.

(b) Konvergenca poteka bolj v smeri podpore nerelacijskih konceptov v relacijskih sistemih, kjer se nerelacijski koncepti dodajajo predvsem kot dodatki (vtičniki).

(c) Sistemi NewSQL predstavljajo sintezo obojih sistemov, a sistemi NewSQL niso rezultat konvergence relacijskih in nerelacijskih sistemov, ki zmanjša razlike med poloma, temveč produkt, ki vsebuje omejen del relacijskih in del nerelacijskih lastnosti.

V nadaljevanju bomo naredili pregled relacijskih in nerelacijskih baz. Slednje so namenjene bolj specifičnim primerom uporabe, kar jih naredi učinkovite za te primere uporabe, hkrati pa omogočajo masivno horizontalno skaliranje. Velik del performans sistemov je odvisen tudi od tega, kako so podatki particionirani.

2.1 Horizontalno in vertikalno skaliranje

Obstajata dva tipa skaliranja. Pri vertikalnem skaliranju dodajamo vire (procesorska moč, pomnilnik) enemu vozlišču (računalniku) v sistemu [3]. Alternativo vertikalnemu skaliranju predstavlja horizontalno skaliranje, v katerem obstoječi sistem nadgradimo z dodajanjem vozlišč (računalnikov) [3]. Horizontalno skaliranje omogoča, da nadomestimo zmogljiv in cenovno drag računalnik z množico cenejših, ki skupno lahko prekašajo procesorsko moč enega samega računalnika, a je treba tak sistem pravilno vzpostaviti in z njim upravljati.

2.2 Horizontalno in vertikalno particioniranje

Ker je količina podatkov, ki jih sistemi hranijo, lahko prevelika, da bi jo hranili na enem vozlišču, se sistemi poslužujejo particioniranja. Pri horizontalnem particioniranju se zapisi shranjujejo na različne strežnike glede na vrednost ključa (imenovano horizontalno deljenje podatkov - sharding). Vertikalno particioniranje za razliko od horizontalnega razdeli zapis in nato dele zapisa shrani na različne strežnike [4].

2.3 Relacijski sistemi za upravljanje s podatkovnimi bazami

Relacijske podatkovne baze so se pojavile leta 1970, ko je Edgar Codd predlagal relacijski podatkovni model. Dandanes je relacijski model najbolj uporabljen model za shranjevanje podatkov [5].

Značilnost relacijskih sistemov je, da so podatki organizirani v fiksno oblikovane tabele s stolpci in vrsticami. Vsaka vrstica predstavlja entiteto (npr. oseba), medtem ko stolpec predstavlja atribut te entitete (npr. ime osebe). Stolpec je najmanjša enota podatkov, na katero se lahko sklicujemo. Imeti mora unikatno ime v tabeli in podatkovni tip, ki pove format podatkov. Podatkovna baza običajno vsebuje veliko enolično poimenovanih tabel, ki so med seboj povezane preko atributov,

s katerimi enolično določimo korespondenco med vrsticami.

Relacijski sistemi za upravljanje s podatkovnimi bazami so sistemi, ki implementirajo relacijski podatkovni model. Izvedejo predstavitev tabel in hranjenje podatkov, definirajo poglede (views), prožilce (triggers), shranjene procedure (stored procedures), omogočajo izgradnjo indeksov, poizvedovanje po podatkovni bazi, upravljanje s podatki in s podatkovno bazo, upravljanje s pravicami dostopa do podatkov in tabel, porazdeljevanje podatkov po različnih strežnikih, zagotavljanje izvajanj ACID transakcij in ostale napredne možnosti.

2.3.1 Komercialni in odprtokodni relacijski SUPB

Na trgu imamo mnogo relacijskih SUPB. Ločimo jih na komercialne in odprtokodne izdelke. Razlikujejo se v ceni in lastnostih, ki jih ponujajo. Med najbolj znane in uporabljene komercialne izdelke spadajo Oracle 11g (12c) [6], IBM DB2 [7] in Microsoft SQL Server [8]. Za komercialnimi sistemi stojijo velika podjetja, ki zagotavljajo kakovost svojih izdelkov. Sistemi ponujajo orodja za upravljanje s podatki, replikacijo in obnovo baze, napredne transakcijske lastnosti in ostalo. Alternativo komercialnim sistemom predstavljajo odprtokodni sistemi, kot so MySQL [9], MariaDB [10] in PostgreSQL [11]. Le-ti so na voljo brezplačno in veliko ponujajo. V tem magistrskem delu se bomo osredotočili na odprtokodne relacijske sisteme MySQL, MariaDB in PostgreSQL.

2.3.2 SQL

SQL je enostaven, ukazni in univerzalni jezik za relacijske podatkovne baze in ga uporablja skoraj vsak relacijski SUPB. Stavke SQL lahko razdelimo v štiri skupine [12, stran 111]:

1. **Data Query Language - DQL** - Omogoča poizvedovanje po podatkih z uporabo stavka SELECT. Stavki ne spreminjajo podatkov v bazi.

2. **Data Manipulation Language - DML** - Omogoča spreminjanje in brisanje zapisov v tabelah preko stavkov INSERT, DELETE in UPDATE.

3. **Data Definition Language - DDL** - Omogoča opredelitev in spreminjanje strukture podatkovne baze (ustvarjanje, spreminjanje in brisanje atributov,

tabel in drugo) preko stavkov CREATE, ALTER in DROP.

4. **Data Control Language - DCL** - Omogoča upravljanje privilegijev uporabnikov podatkovne baze preko stavkov GRANT in REVOKE.

Pomembni mejniki v zgodovini SQL:

- 1982 - Izdan SQL s strani IBM.
- 1986 - Prvi standard SQL-86.
- 1989 - SQL standard gre skozi manjšo revizijo, kjer je SQL dograjen z integritetnimi omejitavmi.
- 1992 - SQL gre skozi večjo revizijo, v kateri se obseg standardne specifikacije petkrat poveča v primerjavi z različico SQL-89. V SQL-92 je dodana podpora novim podatkovnim tipom (DATE, TIME, ...), operacijam (združevanje nizov, nove operacije stikov, ...), spreminjanju sheme z ukazoma ALTER in DROP, preverjanjem integritete z ukazom CHECK in ostalim.
- 1999 - Standard SQL-92 je nadgrajen s standardizacijo prožilcev, regularnimi izrazi in ostalimi funkcionalnostmi v različici SQL-99.

SQL standard je vsesplošno sprejet in je vsaj v neki podmnožici standarda prisoten v vseh relacijskih SUPB.

Da aplikacije lahko uporabljajo SQL in jim pri tem ne bi bilo treba implementirati načina dostopa do baze in načina poizvedovanja (ki sta različna od baze do baze), je postavljen standard ODBC (Open Database Connectivity). ODBC je standardni aplikacijski vmesnik, preko katerega lahko aplikacije komunicirajo z različnimi podatkovnimi bazami. Da je to mogoče, morajo ponudniki SUPB razviti ODBC gonilnike, s katerimi implementirajo povezavo z njihovo podatkovno bazo. ODBC nato pretvori zahteve SQL v zahteve, ki jo posamezen SUPB razume.

2.4 Nerelacijski sistemi za upravljanje s podatkovnimi bazami - NoSQL

Sistemi NoSQL (Not Only SQL) so porazdeljene (horizontalno skalabilne) nerelacijske baze, namenjene hranjenju in vzporedni obdelavi nenehno večje množice

podatkov, imenovane Big data, skozi množico strežnikov [13, stran 1]. Termin Big data se nanaša na zbirko podatkov, ki je postala tako velika, da je ni mogoče učinkovito upravljati s klasičnimi relacijskimi SUPB [13, stran 1]. Običajno gre za količino podatkov reda vsaj nekaj terabajtov [14, stran 7]. Sistemi NoSQL niso novost, a razvoj in nastanek priljubljenih shramb NoSQL je prišel s pojavom interneta in velike količine nestrukturiranih podatkov. Pospešitev popularnosti NoSQL so povzročile strani Google, Amazon, Facebook in druge, kjer uporabniki, sistemi in senzorji ustvarjajo velike količine podatkov [13, stran 2], hkrati pa so ustvarjeni podatki med seboj prepleteni, kompleksni in pridobljeni v različnih oblikah iz množice različnih virov.

Sistemi NoSQL delujejo na nestrukturiranih ali delno strukturiranih podatkih. Porazdeljena zasnova jih naredi idealne za masivno skupinsko procesiranje [13, stran 4]. Delimo jih v štiri večje skupine glede na njihov tip:

1. **Shrambe tipa ključ-vrednost (Key-Value stores).**

   Podatki so shranjeni v obliki ključ-vrednost. Primer preproste programske strukture, ki omogoča tako hranjenje, je preslikovalna tabela (Hash Table, Hash Map). Struktura omogoča dostop do podatkov v konstantnem času $O(1)$ [14]. Odlika teh shramb je izjemna hitrost pri pridobivanju podatkov in horizontalnem skaliranju. Shrambe so učinkovite pri upravljanju spletnih sej, hranjenju sprememb na spletnih straneh [4] (nakupovani voziček) in v podobnih primerih. Nekatere priljubljene shrambe so Redis, Memcached, DynamoDB, SimpleDB, Riak, Aerospike in ostale [15].

2. **Dokumentno usmerjene shrambe (Document databases).**

   Namenjene so hranjenju in upravljanju z dokumenti. Dokumenti so lahko zapisani v obliki XML in JSON (JavaScript Object Notation), v preglednicah in ostalih oblikah. Za razliko od shramb tipa ključ-vrednost so podatki v vrednosti polstrukturirani [13, stran 5]. Vsak stolpec ima lahko več atributov, pri čimer se tip in število atributov lahko spreminjata z vsakim dokumentom [13, stran 5]. Uporabljajo se za različne zbirke dokumentov od besedilnih dokumentov do poštnih sporočil in XML [13, stran 6]. Dokumentne shrambe so uporabne, kadar imamo opravka s podatki, ki jih ne želimo hraniti v stolpcih in vrsticah, ali v primeru, ko shema podatkov ni vnaprej natančno določena

in se lahko spreminja od dokumenta do dokumenta. Primeri priljubljenih baz so MongoDB, CouchDB, Couchbase, Amazon DynamoDB, MarkLogic in ostale [16].

3. **Stolpično usmerjene shrambe (Column-Oriented Stores).**

   Stolpično usmerjena shramba je po izgledu najbolj podobna relacijskim bazam. Podatki so shranjeni v ločenih stolpcih (Column family je množica stolpcev, ki so shranjeni skupaj in morda povezani), za razliko od relacijskih baz, kjer so podatki združeni v vrsticah. Taka struktura omogoča večjo učinkovitost, ko imamo opravka z operacijami (agregacija) nad veliko množico podobnih elementov (npr. isti stolpec). Shramba omogoča učinkovito shranjevanje podatkov in poizvedovanje po načinu Map-Reduce. Uporablja se za porazdeljene shrambe, analitične in skupinske obdelave podatkov [13, stran 6]. Stolpične shrambe dosegajo boljše rezultate napram relacijskim podatkovnim bazam, ko želimo računati agregacijo skozi vse vrstice tabele na manjšem številu stolpcev ali pa, ko imamo na voljo vse vrednosti za stolpce, saj lahko tako učinkovito zapišemo novi stolpec brez vpliva na ostale stolpce v vrstici. Primeri priljubljenih baz so Google BigTable, Cassandra, HBase, Accumulo, Hypertable in Sqrrl [17].

4. **Shrambe za delo z grafi (Graph databases).**

   Gre za relacijski graf med seboj prepletenih parov ključ-vrednost [13]. Bazo grafov sestavljajo vozlišča (objekti), povezave (oz. odnosi med vozlišči) in lastnosti povezav (atributi, izraženi kot pari ključ-vrednost) [13, stran 7]. Te shrambe niso najbolj prilagojene za poizvedovanje, vendar jih uporabljamo, kadar nas zanimajo odnosi med podatki in ne podatki sami. Uporabljajo se npr. pri socialnih omrežjih in v primerih, ko uporaba SQL ni najbolj primerna, npr. poišči prijatelje od prijateljev in jih razvrsti po globini poznanstva. Primeri priljubljenih baz so Neo4J, Titan, OrientDB, Sparksee, Giraph in ostale [18].

2.4.1 Komercialni in odprtokodni NoSQL SUPB

Sistemi NoSQL so večinoma odprtokodni sistemi z različnimi licencami [19]. Ker niso standardizirani, kot npr. relacijski sistemi, je za njihovo uporabo potrebno

specifično znanje. To izkoriščajo različna podjetja, ki nudijo komercialno podporo tem produktom v smislu vzpostavljanja, upravljanja in pomoči pri uporabi sistema (DataStax za sistem Cassandra, Pivotal za Redis). Obstajajo tudi komercialni produkti (MarkLogic, BerkleyDB) in storitve, kot Amazon DynamoDB, ki hrani podatke v oblaku, zagotavlja nizko latenco ter omogoča upravljanje dokumentov in podatkov tipa ključ-vrednost [20].

2.4.2 Cassandra

Cassandra je stolpično usmerjen SUPB, ki je primeren v primerih, ko sta potrebni skalabilnost in visoka razpoložljivost brez zmanjšanja učinkovitosti (performans) [21]. Je porazdeljen sistem, ki je zasnovan tako, da zna obvladati velike količine podatkov čez več strežnikov. Verjetno najbolj znani podjetji, ki uporabljata njene storitve, sta Facebook in Twitter.

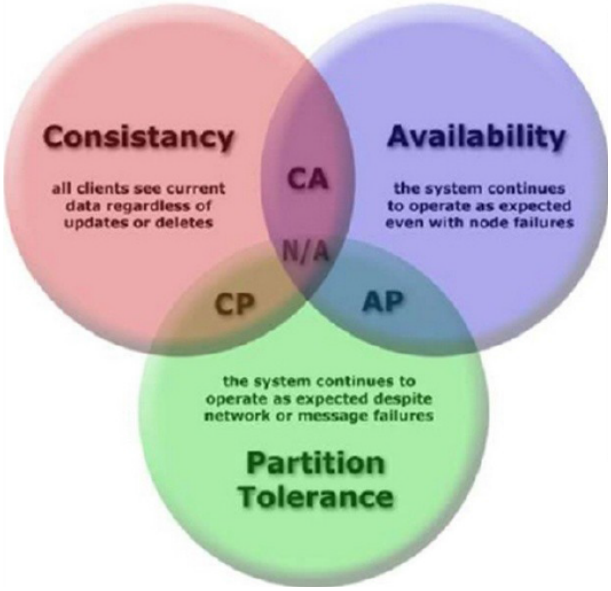
2.5 Dva svetova

2.5.1 ACID in BASE

Relacijski sistemi uporabljajo shemo **ACID (Atomicity, Consistency, Isolation, Durability)**, ki omogoča zagotavljanje skladnosti, integritete, trajnosti in atomarnosti pri izvajanju transakcij brez interakcije uporabnika.

Pomen posameznih elementov ACID je sledeč [14, stran 114]:

- **Atomičnost** - zagotavlja, da se transakcija izvede v celoti ali pa sploh ne (spremembe se razveljavijo).
- **Skladnost** - zagotavlja, da sprememba stanja podatkovne baze povzroči novo skladno stanje, oziroma sprememba podatkov ni dovoljena, če podatki ne ustrezajo prej določeni omejitvi ali pravilu.
- **Izolacija** - zagotavlja, da proces ne more spremeniti dela podatkov, če te podatke uporablja druga operacija v izvajanju.
- **Trajnost** - zagotavlja, da se vse uveljavljene podatke lahko povrne v primeru kakršne koli odpovedi sistema.



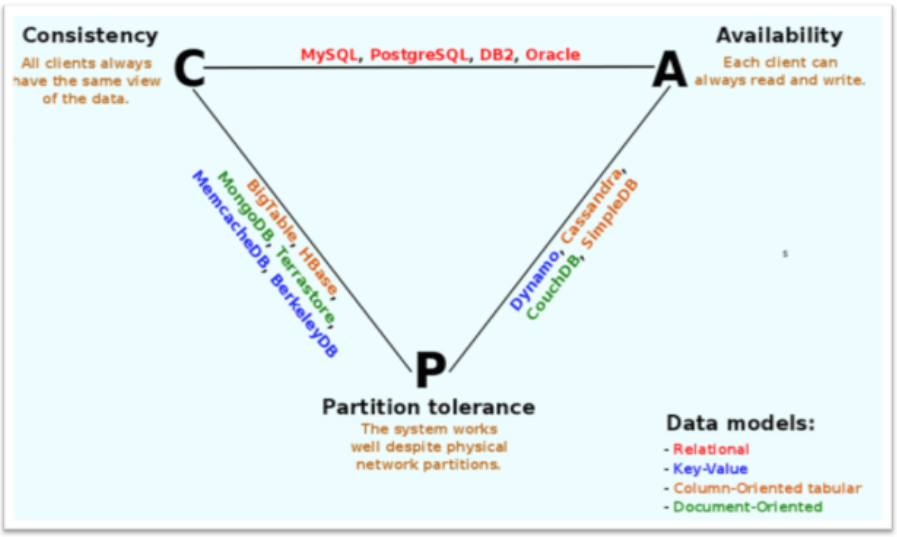
Slika 2.1: CAP teorem. Vir: [13].

Nerelacijski sistemi se odpovedo shemi ACID zaradi razlogov, navedenih v razdelku 2.5.5. Namesto ACID uporabljajo shemo **BASE (Basically Available, Soft State, Eventual Consistency)**, ki se odpove skladnosti podatkov [22].

- **Razpoložljivost** - pomeni, da bo sistem razpoložljiv, ko ga potrebujemo (Basically Available).
- **Mehko stanje podatkov** - pomeni, da se stanje sistema lahko spreminja skozi čas zaradi postopne konsistence (Soft State).
- **Postopna konsistenca** - pomeni, da bo sistem sčasoma prišel v skladno stanje (Eventual Consistency).

2.5.2 CAP

Teorem CAP opisuje, da je na porazdeljenih sistemih mogoče optimizirati dve od treh naštetih lastnosti [22]: particioniranje, razpoložljivost in skladnost [13, stran 3].



Slika 2.2: Umestitev podatkovnih sistemov v CAP teorem. Vir: [23].

2.5.3 Odlike relacijskih SUPB

Odlika relacijskih podatkovnih baz je v strukturiranju podatkov v fiksno oblikovane tabele, možnosti uporabe močnega jezika za poizvedovanje po podatkih (SQL) in uporabi sheme ACID.

V devetdesetih letih prejšnjega stoletja je SQL poenotil sisteme. Je izvedba relacijskega modela in temelji na relacijski algebri, ki matematično zagotovi, da vse baze z uporabo SQL vrnejo isti rezultat, za isto poizvedbo, nad istimi podatki [24]. Vsi relacijski SUPB omogočajo kakšno obliko standarda SQL. Ne glede na izbiro SUPB imamo opravka vsaj s podmnožico standarda ANSI za SQL. Razvijalcem prehod med relacijskimi SUPB ne predstavlja velikih težav, saj se jim ni treba naučiti novega jezika in tehnik dostopa, kajti skupna osnova je SQL.

Učinkovito upravljanje ACID transakcij je ena izmed ključnih lastnosti SUPB, ki jo relacijski sistemi imajo in nekateri NoSQL nimajo. ACID transakcije zagotavljajo, da so relacijske podatkovne baze v skladnem stanju.

Dodatna prednost relacijskih podatkovnih baz je možnost uporabe normalizacije za zmanjšanje neskladnosti med podatki. Normalizacija omogoča, da velike tabele razdelimo na manjše in ustvarimo razmerja med velikimi in malimi tabelami. Podatke spreminjamo (dodajamo, spreminjamo, brišemo) v majhnih tabelah, na-

kar se te spremembe razširijo skozi vse tabele.

Relacijske podatkovne baze so v osnovi namenjene obdelavi OLTP (On-Line Transaction Processing). Uporabljamo jih v sledečih primerih [25]:

- ko upravljamo z enostavnimi operacijami (INSERT, DELETE, UPDATE),
- ko želimo imeti hiter odzivni čas poizvedb,
- ko želimo ohranjati integriteto podatkov,
- ko želimo delati z aktualnimi podatki in
- ko zmogljivost merimo s številom opravljenih transakcij na sekundo.

Relacijske podatkovne baze brez bistvene spremembe v strukturi podatkov niso primerne za analitično obdelavo velikih količin podatkov OLAP (On-Line Analytical Processing).

2.5.4 Omejitve relacijskih SUPB

Relacijski SUPB so zrela in napredna tehnologija, toda s pojavom interneta in nenehno večje količine nestrukturiranih podatkov postajajo v določenih primerih neučinkoviti. Internet je povzročil potrebo po izmenjavi in hranjenju različnih tipov podatkov, od zvočnih in video datotek, kompleksnih tekstovnih in binarnih datotek pa do elektronske pošte in log datotek. Nove zahteve so povzročile, da so nekatere prednosti relacijskih SUPB postale tudi njihove omejitve [26]:

- Struktura relacijske baze je preddefinirana s tabelami in fiksnimi imeni ter tipi stolpcev.
- Horizontalno skaliranje je težko izvedljivo, saj je združevanje tabel (operacija JOIN) v porazdeljenem sistemu zahtevno, relacijski SUPB pa niso zasnovani za deljenje podatkov.
- Ne glede na tip vhodnih podatkov morajo biti vsi podatki pretvorjeni v tabelo. Če pretvorba ni enostavna, je struktura baze lahko kompleksna, težko upravljiva in počasna.

- Poizvedovalni jezik SQL je primeren za obdelavo strukturiranih podatkov, manj pa za delo z nestrukturiranimi.
- Relacijski SUPB ponujajo veliko funkcionalnosti, ki jih uporabniki ne uporabljajo ali ne potrebujejo, a večajo kompleksnost in ceno SUPB.

Glavna kritika relacijskih sistemov je, da niso enostavno skalabilni. Možno jih je vertikalno nadgraditi (z dodajanjem pomnilnika in procesorske moči), a tak sistem postane zelo drag. Relacijske sisteme je sicer možno horizontalno skalirati, vendar z omejitvami, saj je zahtevno vzpostaviti porazdeljeno arhitekturo za sisteme, ki zagotavljajo ACID transakcije in razpoložljivost. Relacijski sistemi to delno rešujejo s horizontalnim deljenjem podatkov. Druga kritika je, da so neučinkoviti pri obravnavi nestrukturiranih podatkov.

2.5.5 Nerelacijski SUPB kot odgovor na pomanjkljivost relacijskih SUPB

Sodobne nerelacijske podatkovne baze so nastale kot odgovor na nekatere pomanjkljivosti (slaba skalabilnost in neučinkovitost pri obravnavi nestrukturiranih podatkov) in nefleksibilnosti klasičnih relacijskih podatkovnih baz, vendar za ceno univerzalnosti, saj pogosto pokrivajo le zelo specifičen razred podatkovnih potreb.

NoSQL so običajno zelo skalabilni sistemi, toda za dosego tega se morajo odpovedati določenim lastnostim, ki so krasile relacijske sisteme. V porazdeljenih sistemih je ACID transakcije težko zagotavljati in sistemi NoSQL imajo pri tem določene omejitve. Odpovejo se stalni skladnosti podatkov, da zagotovijo razpoložljivost in particioniranje podatkov. Večinoma NoSQL sistemi uporabljajo shemo BASE (Basically Available, Soft State, Eventual Consistency), ki se odpove skladnosti.

V veliko primerih ACID transakcije niso potrebne in je postopna konsistentost dovolj. Nekateri NoSQL SUPB ne podpirajo ACID transakcij, nekateri pa jih razvijejo v šibkejši obliki [27]. Velikokrat stroga izolacija (tip serializable) vpliva na sočasnost, kar pomeni, da so izolacijske zahteve velikokrat na nižji stopnji (manj stroga izolacija tipa read uncommited, read commited, repeatable read) [14, stran 171]. NoSQL sistem, ki podpira ACID transakcije, je npr. Neo4j, medtem ko

Redis zagotavlja del ACID (omogoča omejene transakcije, ki so atomične in izolirane, ampak ne zagotavljajo trajnosti in skladnosti) [28]. Sistemi NoSQL ne uporabljajo standardnega povpraševalnega jezika za interakcijo s podatki, temveč vsak sistem uporablja svojo tehnologijo in povpraševalni jezik, kar privede do težave s standardizacijo. Obstaja veliko različnih sistemov NoSQL, poleg tega pa so tudi različnih tipov. Za razliko od relacijskih sistemov podatkov ne hranijo v tabelah, ampak so slednji lahko v dokumentih, grafih, parih ključ-vrednost, tipu JSON in ostalih oblikah. Ker so podatki lahko skorajda v poljubni obliki, je težko ustvariti skupen poizvedovalni jezik, ki bi poenotil upravljanje po teh shrambah. Kljub težavam je del stroke [24] prepričan, da NoSQL potrebuje nekaj takega kot SQL.

Leta 2011 so pri podjetju Couchbase začeli razvijati jezik UnQL (Unified Query Language), ki je požel zanimanje stroke [29]. Projekt je poskušal prenesti značilnosti SQL v domeno NoSQL. Projekt je po enem letu razvoja zaustavil aktivnosti in bil opuščen, vendar razlogi niso natančno znani. Vir [30] kot glavni problem navaja, da je UnQL želel poenotiti poizvedovanje za vse shrambe od tipa ključ-vrednost pa do shramb z grafi, kar je zelo zahtevno in mogoče ni najbolj primerno.

Če je po eni strani smiselno, da NoSQL potrebuje standardizacijo, standardizacija po drugi strani nima smisla. Sistemi NoSQL so bili razviti za specifične primere uporabe, ker so bili ti primeri težko opravljivi z relacijskimi SUPB. Na to nakazuje tudi veliko različnih tipov sistemov NoSQL.

2.6 Konvergenca

S pregledom obeh svetov smo prišli do ugotovitev, da prednosti enega tipa SUPB predstavljajo slabosti drugega tipa SUPB in obratno. S prehodom z relacijskega na nerelacijski SUPB ali obratno bi tako izgubili prednosti predhodnega, zato so začeli relacijski in nerelacijski SUPB nadgrajevati svoje sisteme s koristnimi koncepti, značilnimi za drug tip sistemov. Temu pojavu lahko rečemo kar konvergenca relacijskih in nerelacijskih SUPB. Na tem področju se trenutno veliko raziskuje in razvija. Eden izmed prenesenih konceptov je podpora formatu JSON v relacijskih sistemih [1], ki omogoča enostavno delo z nestrukturiranimi podatki. JSON je format za shranjevane in izmenjavo podatkov ter predstavlja alternativo for-

matu XML. Nekateri avtorji [2] so raziskali področje relacijskih in NoSQL baz ter s preučevanjem prednosti in slabosti ugotovili, da je uporaba izključno enega sistema za vse probleme neprimerna. V okviru magistrskega dela smo identificirali naslednje koncepte.

- PostgreSQL je v različici 9.4 dodal podporo formatu JSON z novim podatkovnim tipom JSONB. JSON se veliko uporablja na spletu in predstavlja osnovo za delo z dokumenti v MongoDB. S tem dodatkom je PostgreSQL razširil funkcionalnosti in ga lahko uporabimo kot dokumentno podatkovno bazo NoSQL, ki konkurira Mongu.

- MySQL je sistem tipa ključ-vrednost Memcached (razdelek 2.4), ki se uporablja za pohitritev operacij s shranjevanjem parov ključ-vrednost, nadomestil z implementacijo svojega internega sistema (API), ki omogoča dostop in hranjenje tipa ključ-vrednost brez uporabe SQL. Podjetje DeNA je izdelalo vtičnik HandlerSocket za MySQL in MariaDB, ki omogoča dostop do podatkov neposredno brez SQL, a v zameno za hitrost zavrže s SQL povezane prednosti, kot so sprožilci in preverjanje vhodnih podatkov (validacija).

- MariaDB je v letu 2014 letu dodala veliko nerelacijskih lastnosti. Razvili so shranjevalni mehanizem, ki omogoča vpogled v tabele sistema Cassandra, povezovanje tabel sistema Cassandra z MariaDB tabelami in poizvedovanje preko SQL. Dopolnili so dinamične stolpce, na katere lahko gledamo kot na alternativo podatkovnemu tipu JSON. Razvili so shranjevalni mehanizem Connect, ki omogoča preslikovanje podatkov, ki so v različnih oblikah (TXT, CSV, TSV, datoteke Excel in druge), v MariaDB tabele, nakar so omogočili povezovanje in manipuliranje teh tabel. Razvit je shranjevalni mehanizem OQGraph, ki omogoča upravljanje z grafi v relacijski podatkovni bazi. Podjetje Tokutek ponuja shranjevalni mehanizem TokuDB za relacijske sisteme, ki naj bi pohitril osnovne operacije in omogočal boljšo kompresijo podatkov kot shranjevalna mehanizma InnoDB in MyISAM.

- Oracle je izdal rešitev za reševanje problema skaliranja, imenovano MySQL Fabric, ki omogoča upravljanje gruče strežnikov, ki uporabljajo shranjevalni mehanizem InnoDB. Prav tako podjetje Netflix [31] razvija rešitev, imeno-

vano Dynomite, ki bi rešila probleme horizontalnega deljenja podatkov za baze, ki niso zasnovane za porazdeljevanje podatkov.

Prihajamo do ugotovitev, da konvergenca poteka bolj v smeri uporabe nerelacijskih konceptov v relacijskih sistemih. Toda konvergenca poteka tudi v drugo smer.

Podjetje FoundationDB je na svoji podatkovni bazi NoSQL tipa ključ-vrednost omogočilo uporabo relacijskega podatkovnega modela in jezika SQL, ki je združljiv s standardom SQL-92 z izjemami (manjka operacija OUTER JOIN, RENAME INDEX ni podprt, READ UNCOMMITTED, READ COMMITTED in REPEATABLE READ izolacijske ravni niso podprte, in še drugo [32]). Omenili smo tudi že poizkus izdelave enotnega poizvedovalnega jezika za NoSQL, imenovanega UnQL, ki pa je opuščen. Obstaja tudi SQL podobna (okrnjena) rešitev GQL (Google Query Language) [33], ki jo je razvil Google za svojo NoSQL shrambo v oblačni storitvi Google App Engine. Nekateri NoSQL, kot sta npr. Neo4j in FoundationDB, podpirajo ACID transakcije, ki so ena izmed glavnih lastnosti relacijskih sistemov.

Konvergenca je povzročila tudi pojav novih oblik baz, t.i. NewSQL, ki poskušajo izkoriščati prednosti obojih, tako relacijskih kot nerelacijskih sistemov. Dandanes sistemi NewSQL še niso razširjeni, nakazujejo pa, da je moč izkoristiti prednosti obeh sistemov. Primeri sistemov so NuoDB, Clustrix, VoltDB, FoundationDB in drugi [34].

2.7 NewSQL kot alternativa

Konvergenca je povzročila nastanek novih tipov baz t.i. NewSQL, ki skušajo izkoristiti prednosti obojih, relacijskih in nerelacijskih podatkovnih baz.

Članek [35] opisuje pomanjkljivosti relacijskih sistemov glede na njihovo zasnovo iz sedemdesetih let prejšnjega stoletja in opiše načine, kako bi pomanjkljivosti lahko odpravili glede na sodobno tehnologijo. Dandanašnja arhitektura je drugačna kot v sedemdesetih letih (drag pomnilnik, poceni disk). Članek kot glavno težavo relacijskih sistemov opredeli dodatno delo pri procesiranju.

- **Upravljanje indeksov** (b-drevesa, hash) in druge sheme zavzamejo veliko procesorskega časa in vhodno-izhodnih enot.

- **Beleženje (logiranje)** predstavlja obremenitev. Tipično relacijske baze zapišejo spremembe v baze in v beležne datoteke, ki se prenašajo na disk za potrebe trajnosti in omogočanje izvajanja ACID transakcij.
- **Zaklepanje tabel in posodabljanje deljenih struktur** zahtevata dodatno delo, saj imamo opravka z več nitmi.
- **Upravljalec medpomnilnika (buffer manager)** povzroči dodatno delo, saj določa, kateri podatki iz diska bodo v nekem času v pomnilniku.

Vse naštete lastnosti, povzete iz članka [35], povzročajo, da se relacijske baze težko skalirajo. Večino dodatnega dela, opisanega zgoraj, povzroči povezava z diskom. Dostop do diska je časovno potraten, zato je bilo razvito večnitenje, ki je omogočilo, da ena nit čaka na rezultate iz diska, medtem ko druga opravlja drugo delo. Da si niti ne prepišejo podatkov, si zaklepajo tabele in upravljajo z deljenimi strukturami. Ker vsi podatki ne gredo v pomnilnik, je bil razvit upravljalec medpomnilnika, ki prenaša podatke v pomnilnik.

Članek [35] predlaga načine za izboljšanje performansa. Pomnilnik dandanes ni tako drag in vanj lahko spravimo cele podatkovne baze. Če bi bili vsi podatki v pomnilniku, bi lahko večnitenje nadomestili z eno nitjo, saj bi bile operacije samo v pomnilniku in zato hitre. Uporaba ene niti bi odpravila potrebo po zaklepanju in upravljanju z deljenimi strukturami. Dodatno ne bi bilo več potrebe po upravljalcu medpomnilnika. Če bi bili podatki v nedeljeni arhitekturi, bi bil sistem zelo skalabilen.

Poglavje 3

Prenašanje nerelacijskih konceptov v relacijske sisteme

V tem poglavlju bomo pregledali nakaj funkcionalnosti NoSQL sistemov, ki so se začele vključevati v relacijske sisteme. Termin "storage engine" bomo v okviru tega magistrskega dela poimenovali shranjevalni mehanizem, toda zavedati se je treba, da je ta več kot le mehanizem za shranjevanje. Je komponenta, ki jo SUPB uporablja za nizkonivojsko izvajanje operacij CRUD (Create, Read, Update, Delete). Komponenta določa, kako so podatki fizično organizirani, definira različne načine indeksiranja, nadzoruje način dostopa in posodabljanje, definira podprte podatkovne tipe, upravlja z zaklepanjem in po možnosti zagotavlja ACID transakcije. V okviru nadaljnjih poglavij bomo omenjali shranjevalne mehanizme Cassandra, Connect, TokuDB, InnoDB in XtraDB. XtraDB je nadgradnja shranjevalnega mehanizma InnoDB, ki vsebuje popravke ter izboljša performanse in skalabilnost napram InnoDB [36]. Od MariaDB različice 10.0.9 je privzet shranjevalni mehanizem.

V nadaljevanju si bomo pogledali NoSQL koncepte, ki so se začeli prenašati v relacijske sisteme MariaDB, PostgreSQL in MySQL.

3.1 Shranjevalni mehanizmi

3.1.1 Shranjevalni mehanizem Connect

Connect (Connect storage engine) je shranjevalni mehanizem, namenjen preslikovanju podatkov iz različnih virov (CSV, TSV, XML, tabel MySQL, beležnih datotek in drugih) in formatov v MariaDB tabele. Novo nastale tabele lahko nato med seboj združujemo, povezujemo z obstoječimi tabelami in z njimi manipuliramo (vstavljamo, brišemo, spreminjamo podatke).

Uveden je bil v različici MariaDB 10.0, z namenom, da bi na enoten način (z SQL) upravljali različne podatke (strukture podatkov), ki so nastali v različnih časih z različnimi tehnologijami, npr. če želimo povezati podatke iz beležnih datotek (tekstovne datoteke) s transakcijami, ki se nahajajo v tabelah v bazi [10]. S tem mehanizmom naj bi odpravili ročno zakodirano logiko, ki je bila doslej potrebna za povezovanje različnih virov.

3.1.1.1 Problematika

Dandanes podjetja veliko podatkov še vedno hranijo v različnih datotekah (večinoma nerelacijskih formatih). Podatki so v datotekah Excel, CSV, TSV, XML, tekstovnih datotekah in ostalih. Običajno se ti podatki preko procesa ETL (extract-izvleči, transform-preoblikuj, load-naloži) naložijo v podatkovno bazo za nadaljnjo obdelavo. Postopki običajno zahtevajo veliko časa in virov, zato se je pokazala potreba po drugačnem načinu dostopanja in upravljanja s podatki, ki niso v podatkovni bazi [37].

Drugi način upravljanja s podatki se imenuje MED (Management of External Data), ki upravlja s podatki, ki niso shranjeni v SUPB, kot da bi bili shranjeni v tabelah. Pojavil se je standard ISO/IEC 9075-9:2003, ki opisuje, kako realizirati in uporabiti MED v SQL tako, da bi z zunanjimi datotekami upravljali preko stavkov SQL. MariaDB ne podpira standarda, ampak je razvila shranjevalni mehanizem Connect, ki MED podpira na enostaven način [37].

3.1.1.2 Izvedba

Ideja shranjevalnega mehanizma Connect je, da ustvarimo tabelo, s katero opišemo zunanjo datoteko. Tabela je lahko notranja (inward) ali zunanja (outward) [38].

Notranjo tabelo definiramo tako, da pri stavku CREATE TABLE ne navedemo, iz katere datoteke bomo brali podatke. Po izvedbi ukaza se ustvari zunanja datoteka, ki jo napolnimo preko operacij INSERT. Brisanje notranje tabele (DROP TABLE) **izbriše** podatke tudi v datoteki.

Zunanjo tabelo ustvarimo, če pri ustvarjanju tabele navedemo obstoječo datoteko. Podatke dodajamo, spreminjamo in brišemo preko ukazov INSERT, UPDATE in DELETE, medtem ko operacije ALTER, DROP in CREATE ne spreminjajo strukture. Brisanje zunanje tabele (DROP TABLE) **ne** izbriše zunanje datoteke.

Pri povezavi tabele z datoteko se podatki **ne** prenesejo v podatkovno bazo. Podatke v datotekah je mogoče indeksirati, pri čemer se v imeniku, kjer je podana izvorna datoteka, ustvari datoteka z indeksi. Pridobivanje podatkov z indeksi [39] je zelo hitro, a počasno, ko se podatke posodobi. Če se ena indeksirana vrednost spremeni, se celoten indeks ponovno zgradi. Indeksi se ne posodobijo, če so datoteke spremenjene neposredno (ne preko tabel), zato morajo biti ročno posodobljeni. Najhitrejše iskanje pri tekstovnih datotekah (TXT, CSV, TSV) dobimo, če so podatki urejeni po stolpcu [40].

Glavne značilnosti shranjevalnega mehanizma Connect so [37]:

- ni potrebe po razširitvah jezika SQL;
- vgrajene ovojnice (embedded wrappers) za veliko zunanjih podatkovnih tipov (datotek in drugo);
- možnost branja in pisanja v zunanje datoteke in vire podatkov;
- pri zunanjemu skeniranju se vrnejo le stolpci, ki so bili uporabljeni;
- podpora indeksom, posebnim in virtualnim stolpcem;
- vzporedna izvedba particioniranih tabel;
- možna izvedba kompleksnih poizvedb na oddaljenih strežnikih;

- vsebuje API, ki omogoča pisanje ovojnic v jeziku C++.

Shranjevalni mehanizem Connect je novost, ki se je pojavila z različico 10. Je v začetku razvoja in v prihodnje bo znano, koliko bo uporaben. Podrobna predstavitev shranjevalnega mehanizma Connect je opravljena na konferenci Fosdem 2014 [41].

3.1.2 Shranjevalni mehanizem Cassandra

Shranjevalni mehanizem Cassandra (Cassandra storage engine) je mehanizem, ki omogoča dostop do podatkov v gruči sistemov Cassandra. Uveden je bil v različici MariaDB 10. Shranjevalni mehanizem ponuja vpogled v gručo sistemov Cassandra ter aplikacijam omogoči branje in pisanje podatkov v stolpce sistema Cassandra preko SQL, kot da bi bila relacijska tabela. Glavni cilj je integracija podatkov med relacijskim in nerelacijskim svetom [42].

3.1.2.1 Izvedba

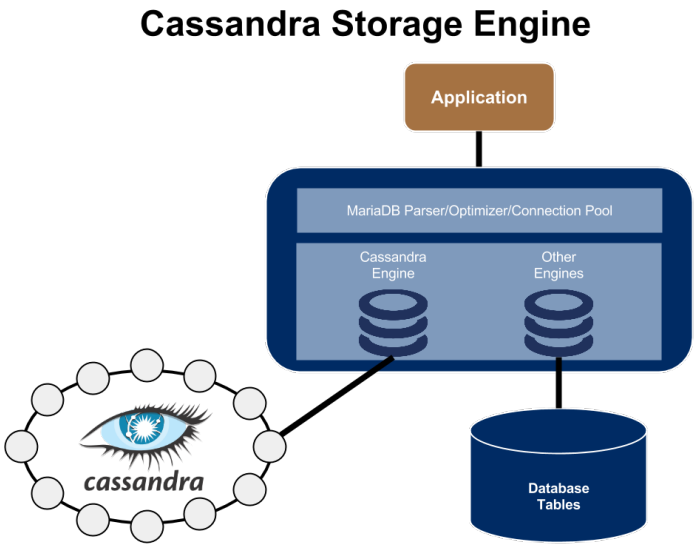
Tabela v MariaDB predstavlja družino stolpcev (Column family) v sistemu Cassandra. Shranjevalni mehanizem prikaže zgradbo sistema Cassandra kot navadno tabelo v MariaDB. Ime tabele v MariaDB je lahko poljubno, a primarni ključ, imena stolpcev in tipi se morajo ujemati s sistemom Cassandra. Cassandra podpira dinamične družine stolpcev, preko katerih lahko dostopamo preko dinamičnih stolpcev. Nepravilna preslikovanja javijo napako [42].

3.1.2.2 Prednosti

Cassandra uporablja povpraševalni jezik CQL, ki je navidez podoben okrnjeni različici SQL, ki ne podpira stikov, podpoizvedb, združevanj (GROUP BY) in urejanj (ORDER BY).

Z vpeljavo shranjevalnega mehanizma Cassandra lahko tabele obdelujemo s SQL. Pregledujemo lahko stolpce ali tabele, izvajamo kompleksne pogoje WHERE in opravljamo operacije stika med tabelami sistema Cassandra in tabelami MariaDB [43].

Če vsaka od instanc MariaDB uporablja shranjevalni mehanizem Cassandra, potem lahko te instance sočasno dostopajo do iste gruče sistemov Cassandra [42].



Slika 3.1: Uporaba shranjevalnega mehanizma Cassandra iz sistema MariaDB. Vir: [10].

Po opisu shranjevalnega mehanizma Cassandra v [42] shranjevalni mehanizem ni primeren za analitične obdelave ogromne količine podatkov, ampak je namenjen kot "okno" za pogled v NoSQL okolje.

3.1.3 Shranjevalni mehanizem TokuDB

TokuDB je odprtokodni shranjevalni mehanizem za MySQL in MariaDB, ki izboljša skalabilnost in učinkovitost teh dveh sistemov [44]. Gre za shranjevalni mehanizem, ki naj bi učinkovito obvladoval z veliko množico podatkov. Namenjen je aplikacijam, ki delajo na masovnih podatkih in imajo težke zahteve, ki jih trenutni shranjevalni mehanizmi ne obvladujejo dobro [44]. Tokutek je v letu 2013 izdal TokuDB Community Edition za MariaDB in MySQL, ki je za razliko od predhodnikov odprtokoden in izdan pod licenco GPL-2.0 [45].

Cilji TokuDB so nadomestiti shranjevalni mehanizem InnoDB, zamenjati B-drevesa s fraktalnimi drevesi [46], izboljšati kompresijo podatkov in pohitriti osnovne operacije (vstavljanje, brisanje, iskanje).

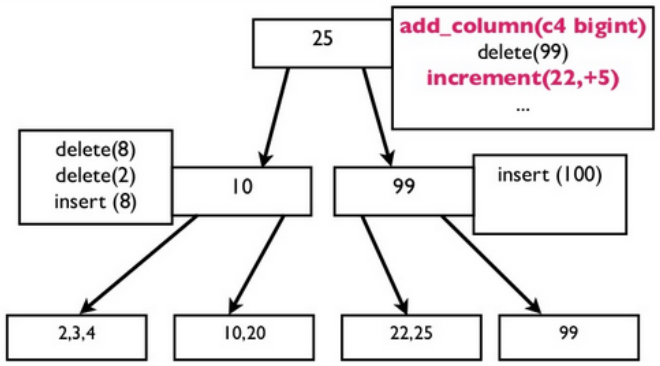
3.1.3.1 Problem B-dreves

Shranjevalna mehanizma InnoDB in MyISAM uporabljata B-drevo za organizacijo podatkov [46]. B-drevesna struktura je zelo hitra, ko gredo vsi podatki v pomnilnik. Ključni podatki so shranjeni v listih drevesa, zato se je treba pri iskanju sprehoditi čez drevo do listov. Sprehod je hiter, ko so vsi podatki v pomnilniku, in počasen, ko podatki ne gredo v pomnilnik. Takrat je za pridobitev podatkov potrebna vhodno-izhodna operacija, ki je časovno zelo zahtevna. Ko podatki ne gredo v pomnilnik, je hitrost B-drevesa omejena na hitrost vhodno-izhodnih enot in diska. Če disk zmore nekaj sto vhodno-izhodnih operacij na sekundo, potem lahko vstavimo zgolj nekaj sto zapisov v drevo in skoraj vsako vstavljanje zahteva vhodno-izhodno operacijo.

3.1.3.2 Fraktalna drevesa

TokuDB uporablja fraktalna drevesa, ki so podobna B-drevesom s ključno razliko [46]. Vmesna vozlišča poleg vozlišč in kazalcev na podvozlišča vsebujejo medpomnilnike (buffers). Po drevesu se pošiljajo sporočila (operacije INSERT, DELETE, UPDATE, BROADCAST, ...), ki se shranijo v medpomnilnik vozlišča. Dokler je v medpomnilniku še prostor, se vanj shrani sporočilo, ko pa medpomnilnik postane poln, se opravi preliv medpomnilnika na podvozlišča (na potomce). Če so medpomnilniki v podvozlišču polni, se rekurzivno povzroči preliv naprej.

Medpomnilniki skladiščijo pisalne operacije. Fraktalno drevo naj bi dosegalo boljše rezultate pri pisanju, ker zanj potrebuje manj vhodno-izhodnih operacij, ki so drage. Pri pisanju v B-drevo opravimo vhodno-izhodno operacijo, v kateri zapišemo eno vrstico, dokument ali vrednost ključ-vrednost. Pri fraktalnih drevesih zapisovanje povzroči preliv (s katerim želimo sprostiti medpomnilnik), kar pomeni, da ena vhodno-izhodna operacija zapiše več vrstic ali dokumentov. S tem se zmanjša število vhodno-izhodnih operacij in odpravi glavni problem B-drevesa.



Slika 3.2: Fraktalno drevo za razliko od B-drevesa v vmesnih vozliščih vsebuje medpomnilnike, ki hranijo sporočila (INSERT, DELETE, UPDATE in ostala). Vir: [47].

3.1.3.3 Kompresija

V TokuDB je kompresija vedno vključena [48] in stisne vse podatke na disk vključno z indeksi [49]. TokuDB stisne velike bloke podatkov reda več MB in s tem doseže dobro kompresijo. InnoDB za razliko od TokuDB stisne bloke velikosti 16 KB. Dodatna prednost TokuDB naj bi bila, da se učinkovitost operacij ne poslabša, če se izvaja kompresija.

3.1.3.4 Hitro spreminjanje sheme (dodajanje stolpcev) - Hot column addition (HCAD)

Spreminjanje sheme tabel v MySQL, npr. dodajanje stolpca, je dolgotrajno. V času spreminjanja sheme se tabela zaklene in s podatki ni moč upravljati. V primeru dodajanja stolpca v tabeli s 100 000 000 zapisi ta operacija lahko traja nekaj ur.

TokuDB odpravi to slabost z vpeljavo t.i. Hot Column Addition (HCAD) [50], ki več ur čakanja nadomesti s sekundami ali minutami. Čas nedostopnosti InnoDB je sorazmeren z velikostjo baze, medtem ko je nedostopnost TokuDB odvisna od časa, ki ga potrebuje SUPB MySQL, da zapre in ponovno odpre tabelo.

Na operacije v TokuDB [51] na tabeli ali indeksu lahko gledamo kot na sporočilo, ki je lahko INSERT, DELETE ali UPDATE. To sporočilo se ne dostavi v trenutku,

ampak se urejeno združuje v medpomnilnikih in potuje proti listom. Ko je operacij dovolj, da se vhodno-izhodna operacija splača, se operacije v pravem vrstnem redu uveljavijo, da se ohrani semantika ukazov SQL.

Ukaz HCAD tvori sporočilo tipa broadcast [51], ki se pošlje vsem vrsticam. MySQL odpre in zapre tabelo na vsak ukaz tipa ALTER. Spreminjanje vrstic se ne zgodi takoj. Nova prihajajoča sporočila potiskajo sporočila tipa broadcast navzdol proti listom. Ko pride na vrsto operacija SELECT, se mora le-ta za razliko od ostalih operacij izvesti takoj. To pomeni, da poizvedba SELECT potisne sporočilo tipa broadcast do ravni, da se sprememba sheme uveljavi. Ko poizvedba doseže vrstico, se le-ta prepiše z dodanim ali odstranjenim stolpcem. Po dodanem stolpcu HCAD ne počne nič več s to vrstico. Ostale neobdelane vrstice med tem časom nimajo dodatnega ali odstranjenega stolpca. Stolpec se doda na vrsticah, ki jih poizvedba obišče. Če želimo, da se stolpec doda na vseh vrsticah naenkrat, moramo izvesti poizvedbo, ki se dotakne vseh vrstic (npr. SELECT COUNT(*) FROM TABLE), lahko pa počakamo, da se spremembe postopoma uveljavijo v ozadju.

3.1.3.5 Več gručnih indeksov - Multiple clustering indexes

Več gručnih indeksov (multiple clustering indexes) pohitri veliko poizvedb. V TokuDB lahko definiramo več gručnih indeksov [52]. Gručni index je indeks, ki hrani vse podatke za vrstico in večina shranjevalnih mehanizmov podpira največ en tak indeks na tabelo (v InnoDB je to primarni indeks) [52]. TokuDB podpira več takih indeksov, kar pomeni boljšo učinkovitost, toda tudi večjo porabo prostora. Gručni indeks je dejansko kopija tabele z različnim urejevalnim zaporedjem.

3.1.3.6 Zakasnitev pri replikaciji in hitro dodajanje indeksov

Zakasnitev pri replikaciji (slave lag) je razlika med časom, ko se operacija izvede na glavnem vozlišču (gospodar), in časom, ko se operacija izvede na drugem vozlišču (suženj). Pri TokuDB trdijo, da so se zakasnitve znebili [53]. TokuDB omogoča t.i. hitro dodajanje indeksov (hot indexing). Ko dodajamo nov indeks, InnoDB zaklene tabelo in ponovno gradi vse indekse [54], pri čemer lahko gradnja indeksa zavzame več ur. Hitro dodajanje indeksov v TokuDB omogoča, da se sočasno ob gradnji indeksa lahko po tabeli poizveduje in v tabelo vstavlja. Gradnja indeksa

lahko traja od nekaj sekund do nekaj minut.

3.1.3.7 Skalabilnost

TokuDB omogoča visoko skalabilnost pri pisanju in brisanju v primerih, ko količina podatkov preseže velikost pomnilnika [55]. Ob zapolnitvi pomnilnika fraktalna drevesa povzročijo preliv medpomnilnikov in s tem izvedejo množico operacij na vhodno-izhodno enoto. To povzroči krajše čase spreminjanja indeksnega drevesa, kar pomeni, da lahko ohranjamo večje število indeksov za prihajajoče podatke. TokuDB deluje dobro pri poizvedbah, ko le-te presežejo velikost pomnilnika, in slabo, ko poizvedbe ne presežejo velikosti pomnilnika [56].

3.1.3.8 Povzetek

Fraktalna drevesa [45] ohranjajo podatke urejene ter omogočajo iskanje in zaporeden dostop v istem času kot B-drevo, toda vstavljanja in brisanja so hitrejša. Sporočila so lahko posredovana na tak način, da so spremembe sheme lahko narejene med operativnim delovanjem baze in v ozadju. Več indeksov se lahko ohranja z enako učinkovitostjo. Dodajanje podatkov v indeks se za razliko od B-dreves dobro odreže.

Nekatere glavne lastnosti naj bi obsegale [44]:

- Do dvajsetkratna pohitritev pri vstavljanju in indeksiranju. Velika učinkovitost (hitrost) se doseže tudi, ko so tabele prevelike za pomnilnik.
- Do 90 % boljša kompresija na disku.
- Hitro spreminjanje sheme (dodajanje stolpcev) med operativnim delovanjem baze.
- Hitro dodajanje indeksov.
- Ni problemov s fragmentacijo.
- Ni zakasnitev pri replikaciji.

3.1.4 Shranjevalni mehanizem OQGraph

Open Query Graph je shranjevalni mehanizem, ki ga je uvedla MariaDB. Različica 2 je bila uvedena v MariaDB 5.2, različica 3 je bila uvedena v MariaDB 10. Z OQGraph so dodali podporo za delo z grafi v relacijski podatkovni bazi, kar sicer omogočajo nerelacijski SUPB, npr. Neo4j [57].

3.1.4.1 Izvedba

OQGraph omogoča obvladovanje hierarhij in kompleksnih grafov [58]. Ima drugačno fizično arhitekturo kot klasična shranjevalna mehanizma InnoDB in MyISAM. Izgled daje uporabniku občutek, da operira s tabelami, toda v ozadju ni tabel.

Za samo delovanje OQGraph je obvezno treba definirati dve tabeli. Ena (v našem primeru connections) služi za vnos povezav med vozlišči, medtem ko je druga (v našem primeru fix_graph) uporabljena za obdelavo in prikaz rezultatov. Omenjeni tabeli sta med seboj povezani.

Izvedbo OQGraph bomo prikazali skozi primer. Najprej definiramo dve tabeli. Prva (actors) hrani podatke o igralcih, druga (connections) je povezana s tabelo actors in hrani povezave med vozlišči (igralci) preko stolpcev node1 in node2. Tabela actors za delovanje OQGraph ni potrebna, je pa obvezna tabela connections, ki ima lahko poljubno strukturo, a mora obvezno vsebovati dve polji (tipa INT), ki ne smeta biti NULL, preko katerih tabelo connections povežemo s tabelo fix_graph (slika 3.4).

Tabela fix_graph

Definicija tabele fix_graph je določena vnaprej in mora biti definirana natanko tako, kot je opisano na sliki 3.4. Kakršna koli sprememba (dodajanje stolpca, spreminjanje tipa stolpca) povzroči napako. Opis parametrov je sledeč.

- **latch** - Opredeljuje algoritem, ki bo uporabljen. Možno je uporabiti v naprej pripravljene algoritme, kot npr. dijkstra, ki poišče najkrajšo pot med vozliščema origid in destid.
- **origid** - Izvorno vozlišče.
- **destid** - Ponorno vozlišče.
- **weight** - Teža vozlišča (privzeta teža je 1).

```
CREATE TABLE mariadb_test.actors(
id INT UNSIGNED NOT NULL,
name VARCHAR(30) NOT NULL,
PRIMARY KEY (id));

CREATE TABLE mariadb_test.connections (
id INT NOT NULL AUTO_INCREMENT,
node1 INT UNSIGNED NOT NULL,
node2 INT UNSIGNED NOT NULL,
PRIMARY KEY(id),
CONSTRAINT tujikljuc1 FOREIGN KEY (node1)
   REFERENCES actors(id),
CONSTRAINT tujikljuc2 FOREIGN KEY (node2)
   REFERENCES actors(id)
);
```

Slika 3.3: Strukturi tabel actors in connections.

```
CREATE TABLE mariadb_test.fix_graph (
latch VARCHAR(32) NULL,
origid BIGINT UNSIGNED NULL,
destid BIGINT UNSIGNED NULL,
weight DOUBLE NULL,
seq BIGINT UNSIGNED NULL,
linkid BIGINT UNSIGNED NULL,
KEY (latch, origid, destid) USING HASH,
KEY (latch, destid, origid) USING HASH)
ENGINE=OQGRAPH data_table='connections'
origid='node1' destid='node2';
```

Slika 3.4: Tabela **fix_graph** je povezana s tabelo connections (**data_table = 'connections'**) in podatke o izvornih in ponornih vozliščih prejema iz stolpcev node1 (**origid = 'node1'**) in node2 (**destid = 'node2'**).

```
select * FROM actors JOIN fix_graph ON (fix_graph.linkid=actors.id) WHERE
latch='dijkstra' AND origid=1 AND destid=8;
+----+--------------------+----------+--------+--------+--------+------+--------+
| id | name               | latch    | origid | destid | weight | seq  | linkid |
+----+--------------------+----------+--------+--------+--------+------+--------+
|  1 | Christian Bake     | dijkstra |      1 |      8 |   NULL |    0 |      1 |
|  3 | Rihard Din Anderson | dijkstra |      1 |      8 |      1 |    1 |      3 |
|  7 | Zoe Salama         | dijkstra |      1 |      8 |      1 |    2 |      7 |
|  8 | Helen Pagy         | dijkstra |      1 |      8 |      1 |    3 |      8 |
+----+--------------------+----------+--------+--------+--------+------+--------+
```

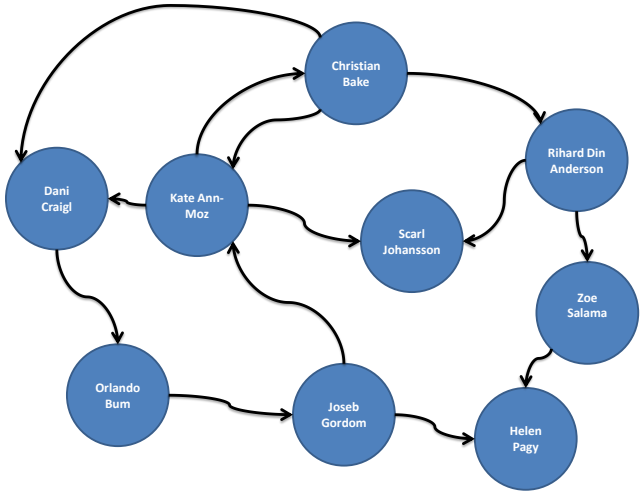
Slika 3.5: Rezultat iskanja najkrajše poti med začetnim vozliščem 1 in končnim vozliščem 8. Pot gre skozi vozlišča 1, 3, 7 in 8.

- **seq** - Števec korakov od izvornega vozlišča do ponornega vozlišča.
- **linkid** - Izpiše vmesno vozlišče, ki je na poti med izvornim in ponornim vozliščem. Je pomembno polje, saj preko njega povezujemo vozlišča z dejanskimi entitetami (v našem primeru actor).

Na koncu definicije so dodani parametri:

- **data_table = 'connections'** - pomeni, da bo tabela fix_graph pridobivala podatke iz tabele connections.
- **origid = 'node1'** - pomeni, da bo tabela fix_graph za izvorno vozlišče pridobivala podatke s stolpca node1 (tabela connections).
- **destid = 'node2'** - pomeni, da bo tabela fix_graph za ponorno vozlišče pridobivala podatke s stolpca node2 (tabela connections).

Na koncu definicije tabele je lahko neobvezno dodan parameter **weight**, s katerim lahko definiramo težo med dvema vozliščema (privzeta teža je 1). Povezave v grafu so usmerjene. V kolikor želimo imeti neusmerjeno povezavo, je treba dodati še povezavo v obratno smer.



Slika 3.6: Primer grafa.

3.1.4.2 Omejitve

OQGraph je optimiziran za preprosta vprašanja, povezana z grafi. Od kod lahko pridem do vozlišča x? Kam grem lahko od vozlišča x? Najdi najkrajšo pot od vozlišča x do vozlišča y. Kolikšen je seštevek teže poti od vozlišča x do vozlišča y? Izpiši vsa vozlišča, iz katerih lahko po neki poti pridemo do vozlišča x, in podobno.

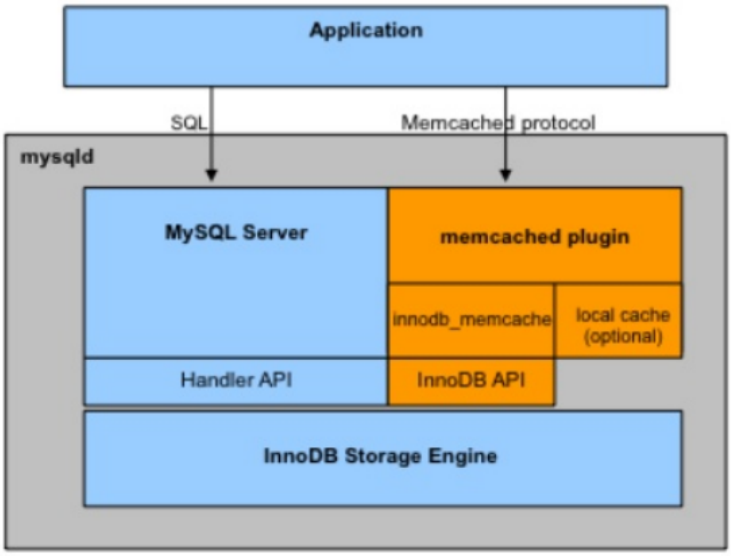
Ni pa OQGraph namenjen za bolj kompleksna vprašanja, povezana z grafi (npr. poišči potencialne prijatelje, ki imajo radi podobne stvari (glej [59])).

Zelo težko je OQGraph primerjati s shrambo Neo4j, saj je bil zasnovan za bolj specifične primere uporabe (uporaba algoritmov na grafih), medtem ko Neo4j omogoča bolj kompleksne poizvedbe.

3.2 Nerelacijske funkcionalnosti

3.2.1 Memcached API

Memcached API je vtičnik za MySQL, ki podatke hrani v posebni tabeli in omogoča pridobivanje le-teh neposredno iz baze mimo plasti SQL preko API klicev [60]. Kot vtičnik v spodaj opisani obliki se je pojavil v različici MySQL 5.6. Namen vtičnika je pohitriti poizvedbe, ki so tipa ključ-vrednost, in razbremeniti vire procesorja,



Slika 3.7: Implementacija Memcached API v MySQL za InnoDB. Vir: [62].

vhodno-izhodnih enot in pomnilnika.

3.2.1.1 Memcached

Memcached [61] je sistem, ki omogoča hitro poizvedovanje po podatkih. Je shramba tipa ključ-vrednost v pomnilniku. Njen namen je hraniti podatke, ki so pogosto v uporabi in pohitriti spletne aplikacije z zmanjšanjem obremenitve podatkovne baze. Ob prejeti zahtevi se v Memcached preveri, ali so podatki že v shrambi. Če so, se jih posreduje v odgovoru. Če podatkov ni, se poizvedba posreduje podatkovni bazi. Odgovor se napolni s podatki iz podatkovne baze in se posreduje odjemalcu, hkrati pa se shramba ključ-vrednost posodobi z novim zapisom. Podatki v predpomnilniku imajo določeno časovno dobo in se po preteku le-te izbrišejo iz shrambe.

3.2.1.2 Izvedba

MySQL v resnici ne uporablja Memcached, ampak njegov API kot vmesnik, ki omogoča pridobivanje podatkov neposredno iz baze mimo plasti SQL. Razlika s sistemom Memcached je v izvedbi. Pri uporabi MySQL rešitve je treba definirati

tabelo, kamor se bodo shranjevali podatki. Definirana tabela je običajna tabela v shranjevalnem mehanizmu InnoDB. To tabelo je treba povezati s konfiguracijskimi tabelami, opisanimi v nadaljevanju (containers, config_options in cache_policies), ki povedo vtičniku memcached, kam naj shranjuje podatke pri uporabi metod get in set (poizvedbe tipa ključ-vrednost). Ker se podatki shranjujejo v navadno tabelo, je moč poleg uporabe get in set uporabiti tudi zahtevnejše poizvedbe s SQL s konstrukti COUNT in GROUP BY. Kot prednost lahko torej izpostavimo, da MySQL rešitev poleg uporabe poizvedb ključ-vrednost omogoča tudi uporabo poizvedb SQL, po drugi strani pa kot slabost navedemo, da je treba definirati strukturo (tabelo), v katero se bodo shranjevali podatki.

Po namestitvi vtičnika se ustvarijo tri tabele (containers, config_options in cache_policies) [63]. Tabela containers opisuje tabele, ki bodo hranile podatke. Tabela cache_policies določa, ali bomo uporabili InnoDB kot shrambo za Memcached, ali bomo uporabili tradicionalni Memcached mehanizem, ali pa bomo uporabili oba (če ključa ni v pomnilniku, se poišče v InnoDB tabeli). Tabela config_options definira ločilnik, ki je omenjen spodaj.

Tabela innodb_memcache.containers sestoji iz sledečih stolpcev, v katerih definiramo nastavitve, ki bodo v uporabi, ko bomo klicali ukaz tipa Object.get(key).

- **db_schema** - Določi podatkovno bazo.
- **db_table** - Določi tabelo, v katero se bodo shranjevali podatki.
- **db_key_columns** - Določi stolpec, v katerega se bo shranil ključ.
- **value_columns** - Določi stolpce, v katere se bodo shranile vrednosti (več stolpcev ločimo z znakom | oziroma z ločilnikom, ki je določen v tabeli config_options).
- **unique_idx_name_on_key** - Ime indeksa na stolpec ključa (PRIMARY, SECONDARY).
- **flags** - Določa, kateri stolpci so uporabljeni kot zastavice (uporabniško definirane številčne vrednosti, ki so shranjene in pridobljene poleg glavne vrednosti; vrednost 0 pomeni, da stolpec ni uporabljen).
- **cas_column** - Čas zamenjave. Vrednost 0 pomeni, da stolpec ni uporabljen.

- **expire_time_column** - Čas preteka. Vrednost 0 pomeni, da stolpec ni uporabljen.

Stolpca cas_column in expire_time_column sta redko potrebna, saj je vtičnik InnoDB memcached tesno integriran v proces in je upravljanje s pomnilnikom upravljano s strani MySQL.

Uporaba Memcached v SQL naj bi omogočala hitrejše poizvedbe in razvojne cikle [62]. Memcached je razvit v mysqlID proces kot vtičnik za shranjevalni mehanizem InnoDB in zagotavlja nizko latenco (teče v istem procesnem prostoru). Obide plast SQL (ni razčljenjevanja) in zagotavlja visoko učinkovitost (hitrost) za poizvedbe tipa ključ-vrednost [62].

Memcached vmesnik pri uporabi ukazov DML (set, incr, ...) zaklepa bodisi vrstice bodisi tabelo (odvisno od konfiguracije), zato sočasna uporaba memcached in SQL (DML, DDL) na istih podatkih ni mogoča [64], [65]. Je pa možno sočasno brati podatke, če je izolacijska stopnja nastavljena na READ UNCOMMITTED.
Nekatere prednosti vtičnika [66]:

- Neposreden dostop do tabel v InnoDB brez razčlenjevanja SQL.
- Ni dodatnega omrežnega zaglavja pri pošiljanju zahtev, saj Memcached teče v istem procesnem prostoru.
- Podatki se transparentno vpišejo in berejo iz InnoDB tabele brez SQL.
- Prenos med pomnilnikom in diskom je avtomatičen.
- Še vedno lahko dostopamo do tabele preko SQL in izvajamo kompleksne operacije.
- Omogoča serializacijo kompleksnih struktur binarnih datotek in blokov kode v nize, ki jih lahko shranimo v bazo.

3.2.2 HandlerSocket

HandlerSocket [67] je vtičnik za MariaDB in MySQL, ki omogoča izvajanje operacij CRUD z neposrednim dostopom do shranjevalnih mehanizmov InnoDB/XtraDB in Spider. Gre za vtičnik NoSQL, ki so ga leta 2010 razvili pri podjetju DeNA za MySQL.

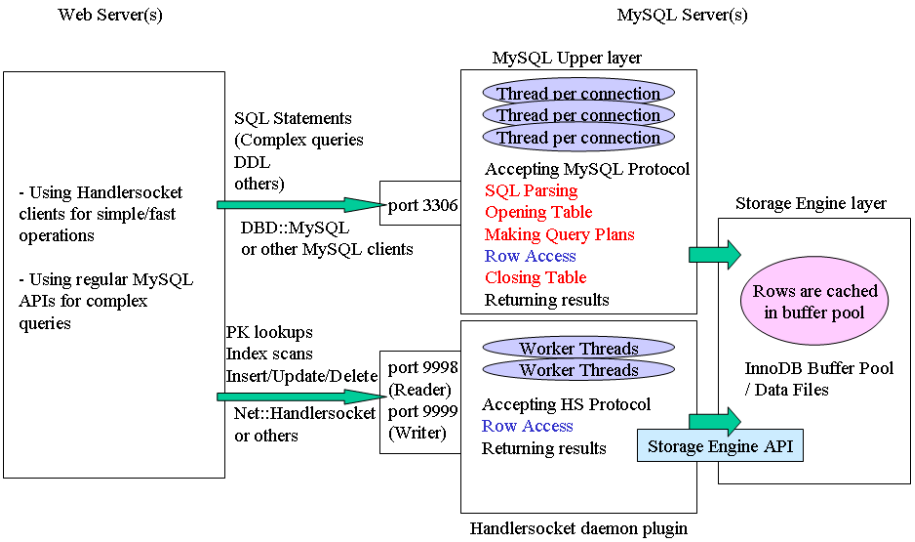
Želja je bila dobiti dostop do podatkov brez SQL, saj naj bi glavni problem uporabe plasti SQL, po ugotovitvah v blogu [68], predstavljalo razčlenjevanje stavkov SQL, odpiranje in zaklepanje tabel in ustvarjanje izvedbenih načrtov SQL. Vse to je časovno portratno, zato so želeli pohitriti osnovne operacije vstavljanja in iskanja ter dodati enostaven in hiter (NoSQL) dostop do podatkov, hkrati pa so želeli omogočati tudi kompleksno poizvedovanje s SQL. Implementirali so vtičnik HandlerSocket, ki naj bi sedemkrat hitrejše izvajal operacije od običajnega dostopa preko SQL [68].

3.2.2.1 Izvedba

HandlerSocket je zasnovan kot vtičnik, ki posluša na specifičnih vratih (9998 za branje, 9999 za pisanje). Sprejema nerelacijske protokole/API in dostopa do shranjevalnega mehanizma neposredno preko notranjega vmesnika. HandlerSocket uporablja svoj protokol in ukaze in nič SQL. Slednjega obide in se tako znebi časovnih presežkov (overhead) pri njegovi uporabi. HandlerSocket ne odpre in zapre tabel za vsako poizvedbo, ampak jih ohranja odprte za ponovno uporabo, kar pomaga izboljšati učinkovitost [68].

Delovanje vtičnika HandlerSocket prikazuje slika 3.8. HandlerSocket posluša na vratih (9998, 9999) z več nitmi (WorkerThreads). Vsaka nit sprejme tisoče povezav in dela z njimi sočasno (za razliko od MySQL, ki dodeli eno nit na povezavo [69]). Za dostop do vtičnika HandlerSocket iz nekega programskega jezika moramo uporabiti za to namenjene knjižnice [70].

Pri branju se pridobi poizvedbe od množice odjemalcev. Sledi zaklepanje tabele in izvršitev množice poizvedb. Po vseh opravljenih poizvedbah se tabela odklene in sočasno vrne odgovore vsem odjemalcem. Tabela se zaklene in odklene enkrat na cikel (količinska obdelava) in ne za vsako poizvedbo. To predstavlja prednost, v določenih primerih pa lahko tudi slabost. Npr. če je v množici enostavnih poizvedb kompleksna poizvedba, se rezultati poizvedb ne bodo poslali odjemalcem, dokler ne bodo vse poizvedbe zaključene. Odjemalec lahko pošilja eno zahtevo naenkrat in dobi nazaj en odgovor naenkrat. Lahko pa pošilja n zahtev naenkrat in dobi nazaj n odgovorov v vrstnem redu zahtev. S primerjalnimi testi (razdelek 6.5) smo prišli do ugotovitve, da uporaba količinske obdelave poizvedb povzroči velikansko povečanje števila opravljenih operacij na časovno enoto.



Slika 3.8: Delovanje HandlerSocket. Vir: [68].

Skušali smo ugotovitvi, ali je mogoče sočasno dostopati do istih podatkov v InnoDB tabeli z vtičnikom HandlerSocket in SQL. Zaklenili smo tabelo in onemogočili pisanje vanjo. Izkaže se, da HandlerSocket obide tudi zaklepanje tabel, saj je bilo v tabelo moč zapisovati. S tem prihajamo do ugotovitev, da je sočasni dostop do podatkov s strani vtičnika HandlerSocket in SQL mogoč. Uporaba vtičnika HandlerSocket lahko pusti podatkovno bazo v nekonsistentnem stanju, saj obide vse s SQL povezane prednosti (opisano v razdelku 3.2.2.3).

3.2.2.2 Primer sintakse

HandlerSocket sintaksa ni enostavno razumljiva za razliko od SQL. Slika 3.9 prikazuje, kako izvedemo osnovno poizvedovanje, podrobni opisi sintakse s primeri pa so navedeni v viru [71]. Za poizvedovanje nad tabelo najprej odpremo indeks in nato izvedemo poizvedbo.

Sintaksa za odpiranje povezave (indeksa)	P <index_id> <db> <table> <index> <columns> [<fcolumns>]
Sintaksa za bralno poizvedbo	<index_id> <op> <vlen> <v1> ... <vn> [LIM] [IN] [FILTER]
	PRIMER
Primer odpiranja indeksa	P 55 test ocene Primary ime,priimek,starost,ocena
Primer izvedbe poizvedbe	55 = 1 3
Rezultat poizvedbe	Vrne stolpce ime, priimek, starost, ocena s podatkovne baze test in tabele ocene, kjer je id = 3
SQL ekvivalent	SELECT ime,priimek,starost,ocena FROM test.ocene WHERE id=3
Ukaza (odpiranje indeksa in poizvedbo) lahko izvedemo preko ukazne vrstice v operacijskem sistemu Linux	echo -e "P\t55 \ttest \tocene\tPRIMARY\time,priimek, starost,ocena\n55\t=\t1\t3 " | nc localhost 9998

Slika 3.9: Primer HandlerSocket sintakse. Z odpiranjem indeksa povemo, katero bazo, tabelo in stolpce bomo uporabili, nakar indeks (število, ki referencira te nastavitve) uporabimo kot referenco pri poizvedovanju. Razlaga sintakse s primeri je v viru [71].

3.2.2.3 Omejitve

HandlerSocket nima podpore SQL, ne podpira shranjenih procedur, agregacij, nekaterih osnovnih operacij (npr. stikov) in podatkovnih tipov, ne podpira aktivacije sprožilcev, pogledov, nima komercialne podpore in protokol se spreminja od časa do časa. Protokol sam po sebi ne zagotavlja varnosti [71] in konsistentnosti. Vsaka nit se izvaja z pravicami sistemskega uporabnika, tako da lahko aplikacije dostopajo do vseh tabel preko HandlerSocket protokola [68]. HandlerSocket ne preverja, ali so vneseni podatki skladni s tipi in omejitvami, ki jih definiramo v shemi.

3.2.2.4 Primerjava z Memcached API

HandlerSocket in Memcached sta si med sabo podobna, saj sta oba razvita z namenom pohitritve operacij z obidenjem plasti SQL. Oba tečeta znotraj MySQL procesa, oba podpirata enostavne operacije in pri obeh je moč na tabelah izvajati operacije SQL. HandlerSocket, za razliko od Memcached API, omogoča ne samo iskanje po ključu, ampak tudi z operandom enakosti in neenakosti (kar lahko vrne

0, 1 ali več vrstic). Kljub njuni podobnosti pa so naši primerjalni testi (razdelka 6.4 in 6.5) pokazali velike razlike med njima. Eden izmed razlogov za te rezultate bi lahko bil v tem, da je HandlerSocket odličen pri količinski obdelavi operacij.

3.2.3 Dinamični stolpci

Dinamični stolpci (Dynamic Columns) [72] so stolpci, ki lahko hranijo poljubno strukturirane podatke in približajo relacijsko shemo MariaDB bližje nerelacijski. Omogočajo namreč, da vsaka vrstica tabele vsebuje različne stolpce (atribute). Prvič so bili predstavljeni v različici 5.3, sedaj pa so izboljšani v različici 10 [10]. Razvili so jih z namenom, da bi dodali podporo hrambi nestrukturiranih podatkov in pri tem ohranjali ACID transakcije [10].

3.2.3.1 Izvedba

Dinamični stolpec združi skupino stolpcev v stolpec tipa BLOB. Stolpec tipa BLOB omogoča shranjevanje strukturiranih podatkov [73]. Sestavljen je iz parov ključ-vrednost, MariaDB pa je uvedla možnost upravljanja s temi stolpci. V različici 10 so omogočili sklicevanje po imenu stolpcev (prej samo po številki) in podprli izvoz podatkov iz dinamičnih stolpcev v format JSON [73]. Dinamični stolpci so primerni za uporabo, kadar ima nek zapis veliko ali neznano število atributov. Na dinamične stolpce lahko gledamo kot na enostaven JSON ali pa hstore [74] zapis. V različici 10 lahko stolpce rezultata po poizvedovanju izpišemo v formatu JSON. Dinamičnih stolpcev ne moremo neposredno indeksirati, toda možno jih je indeksirati s pomočjo virtualnih stolpcev. Virtualni stolpci [75], tudi računalniško generirani stolpci, so stolpci, katerih vrednost se avtomatično izračuna glede na izraz, v večini primerov iz vrednosti drugih stolpcev v tabeli. Lahko so trajni (PERSISTENT) in shranjeni v bazi, ali pa virtualni (VIRTUAL), ki se ustvarijo za vsako poizvedbo na tabelo. Indeksiranje je moč izvesti na trajnih (PERSISTENT) stolpcih [73]. Sledi nekaj primerov uporabe dinamičnih stolpcev.

```
CREATE TABLE oseba (id integer, osebni_podatki blob, izobrazba blob);

INSERT INTO oseba VALUES (1, COLUMN_CREATE('ime', 'Janez', 'priimek', 'Novak'),
COLUMN_CREATE('osnovna_sola', 'Janko Kersnik'));

INSERT INTO oseba VALUES (2, COLUMN_CREATE('ime', 'Ana', 'priimek', 'Horvat', 'starost', 18),
COLUMN_CREATE('osnovna_sola', 'Janko Kersnik'));

UPDATE oseba SET izobrazba = COLUMN_ADD(izobrazba, 'srednja_sola', 'Ledina')
WHERE id = 1;

SELECT id, COLUMN_JSON(osebni_podatki) FROM oseba
WHERE COLUMN_GET(osebni_podatki, 'ime' AS CHAR) = 'Ana';
```

Slika 3.10: Primer uporabe dinamičnih stolpcev.

Na posameznih vrsticah je možno uporabiti sledeče operacije [72]

COLUMN_CREATE (ime stolpca ali številka stolpca, vrednost [AS tip], [ime stolpca ali številka stolpca, vrednost [AS tip]] ...);

Tipa nam ni treba določati, saj zna sistem MariaDB sam obravnavati vnesene podatke, z izjemo v primerih, ko želimo, da se nek znakovni niz obravnava kot datum: npr: '2014-03-01' AS DATE.

COLUMN_ADD (dinamični stolpec ali prazen niz, ime stolpca ali številka stolpca, vrednost [AS tip], [ime stolpca ali številka stolpca, vrednost [AS tip]] ...);

Če stolpec obstaja, se prepiše. Če je vrednost NULL, se stolpec izbriše.

COLUMN_GET (dinamični stolpec, ime stolpca ali številka stolpca AS tip); Če stolpec ne obstaja, se vrne NULL. Potrebno je določiti tip, ki se bere, saj mora tolmač SQL vedeti tipe vseh izrazov pred izvedbo poizvedbe.

COLUMN_DELETE (dinamični stolpec, ime stolpca ali številka stolpca, [ime stolpca ali številka stolpca] ...); Izbriše stolpec.

COLUMN_EXIST (dinamični stolpec, ime stolpca ali številka stolpca); Vrne 1, če stolpec obstaja, v nasprotnem primeru vrne 0.

COLUMN_LIST (dinamični stolpec); Vrne z vejico ločen seznam imen stolpcev.

COLUMN_CHECK (dinamični stolpec); Preveri, ali je dinamičen stolpec pravilno strukturiran blob. Vrne 1, če je, v nasprotnem primeru vrne 0. Uporabimo ga, če se stolpec pretvori iz enega kodirnega sistema v drugega.

COLUMN_JSON (dinamični stolpec); Vrne podatke v obliki JSON. Stolpci, ki so gnezdeni globlje od 10 nivojev, bodo predstavljeni kot binarni niz.

Stolpce je moč gnezditi. Primer: column_create('starš', column_create('otrok', 12345));

Več o podatkovnih tipih, ki so na voljo za stolpce, je opisano v [72].

3.2.4 JSONB

JSONB je podatkovni tip, ki lahko hrani podatke v obliki JSON. Validacija za podatkovni tip JSON se je pojavila v različici PostgreSQL 9.2, nakar so bile v različici 9.3 dodane funkcije nad JSON, podatkovni tip JSONB pa je novost v verziji 9.4. Namen PostgreSQL je bil ustvariti podatkovni tip, ki bo po funkcionalnosti podoben podatkovnemu tipu XML [1].

3.2.4.1 Razlogi za nov podatkovni tip JSONB

PostgreSQL od različice 8.2 podpira format XML, ta se shranjuje v bazo kot tekst in omogoča validacijo XML pri shranjevanju v bazo. Ker je shranjen kot tekst, ne pozna operatorjev za primejave in ekvivalence, tako da tudi ni indeksiranja [76]. Indeksiranje sicer lahko izvedemo tako, da z uporabo XPath (poizvedovalni jezik za navigacijo po elementih in atributih dokumenta XML) dobimo podatke, ki jih nato indeksiramo. Obstaja nekaj funkcij za generiranje XML in predikatov, ki preverjajo, ali je XML veljaven in podobno. XML ima torej dobro lastnost, ker uporablja XPath, toda v večini primerov [77] je poizvedovanje kompleksno (dolga in zapletena koda) in, v kolikor neprimerno izvedemo poizvedbo, tudi dolgotrajno.

Razlogov, da se je PostgreSQL odločil za nov podatkovni tip JSONB, je več. JSON se je uveljavil kot de facto standardni API za REST storitve, omogoča prija-

zno delo z jezikoma Python in Ruby ter je tip, v katerem svoje podatke shranjuje MongoDB [78]. Verjetno glavni razlog za implementacijo podatkovnega tipa je uspešnost podatkovnega tipa hstore (podatkovni tip, ki hrani pare ključ-vrednost). Le-ta je shranjen v binarni obliki in omogoča množico različnih operatorjev za poizvedovanje, indeksiranje in prinaša dobre rezultate [77].

Na format JSON lahko gledamo kot tip hstore z dodanimi možnostmi gnezdenja elementov. Podatkovni tip JSONB je bil razvit na podlagi kode za hstore različico 2 (kodo hstore, ki je omogočala gnezdenje, so prestrukturirali za namen JSONB). Rezultat razvoja je binarni tip JSONB, ki je celotna implementacija formata JSON, ki podpira vse JSON operatorje z nekaj dodanimi hstore operatorji (tabela 3.1), omogoča primerjanje JSON objektov, vsebovanost, eksistenco, omogoča indeksiranje, odstranjuje duplikate ključev napram tipu JSON, omogoča hitre operacije in ostale zanimive lastnosti.

3.2.4.2 JSON in JSONB

JSON se veliko uporablja na spletu, predvsem preko storitev REST. Primeren je v okoljih, kjer so zahteve spremenljive (tekoče). Toda tudi v aplikacijah, kjer je zahtevana najvišja prilagodljivost, je priporočljivo, da imajo dokumenti JSON nespremenljivo strukturo in je s tem lažje pisati poizvedbe, ki povzamejo neko množico dokumentov v tabeli.

Podatkovni tip JSON [79], ki se je pojavil v verziji 9.2, omogoča, da se dokument JSON v podatkovno bazo vstavi v polje tipa tekst, pri čemer se dokument ne predprocesira. V polje se shrani enak dokument, kot ga vpišemo (presledki se ohranijo, lahko obstajajo podvojeni ključi). Preverja se pravilnost dokumenta. Različica 9.3 je dopolnila tip JSON s funkcijami za branje in pripravljanje dokumenta JSON, pretvarjanje v ostale podatkovne tipe in operatorje.

Podatkovni tip JSONB [79] se za razliko od podatkovnega tipa JSON ne obravnava kot besedilno polje. Pred shranjevanjem v podatkovno bazo se predprocesira (presledki se ne ohranjajo in če je v zapisu JSON več ključev z istim imenom, se ohrani zadnji; vrstni red ključev je razporejen po dolžini, nato pa po bitnih primerjavah). Rezultat se v binarni obliki hrani v podatkovni bazi. Shranjen JSONB ne potrebuje ponovnega razčlenjevanja in omogoča indeksiranje. Shranjevanje zapisa v podatkovno bazo traja dlje časa zaradi dodatnih podatkov, ki so potrebni za

```
CREATE TABLE oseba (podatki jsonb);

INSERT INTO oseba (podatki) VALUES ({ "ime" : "Janez", "priimek" : "Novak",
 "šolski_uspehi" : { "odličen" : 3, "prav_dober" : 5 } } );

SELECT FROM oseba (podatki) WHERE (podatki->'šolski_uspehi'->>'odličen')::int = 3;
```

Slika 3.11: Primeri ustvarjanja, vstavljanja in poizvedovanja po tabeli s tipom JSONB.

pretvorbo zapisa v ustrezno obliko. Končna velikost JSONB v primerjavi z JSON je večja.

3.2.4.3 Operatorji in funkcije nad JSON

Vira [80], [79] vsebujeta operatorje, funkcije in primere uporabe. Nekaj pomembnih operatorjev prikazuje tabela 3.1.

Operator	Tip	Opis
`->`	int	Dobi element iz polja.
`->`	text	Dobi vrednost JSON objekta preko ključa.
`->>`	int	Dobi element iz polja kot tekst.
`@>`	jsonb	Ali levi JSON vsebuje desno vrednost?
`<@`	jsonb	Ali je levi JSON vsebovan v desni vrednosti?
`?`	text	Ali ključ/element obstaja?

Tabela 3.1: Pogosto uporabljeni operatorji v JSONB.

Primer	Rezultat
`'[{"a":1},{"b":2}]'::jsonb->1`	`{"b":2}`
`'{"a":{"b":"c"}}'::jsonb->'a'`	`{"b":"c"}`
`'[1,2,3]'::json->>1`	2
`'{"a":1,"b":2}'::json->>'b'`	2
`'{"a":1, "b":2}'::json @> '{"b":2}'::jsonb`	true
`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb`	true
`'{"a":1, "b":2}'::jsonb ? 'b'`	true

Tabela 3.2: Primeri rezultatov uporabe operatorjev v JSONB.

3.2.5 Hstore

Hstore je shramba tipa ključ-vrednost znotraj PostgreSQL. Uporabljamo ga lahko podobno kot slovar v programskih jezikih [74].

V različici 9.4 so pri PostgreSQL želeli izpopolniti podatkovni tip hstore s podporo gnezdenju. Sočasno so razvijali podporo tako JSONB in gnezdenju v hstore, pri čemer je nastala razlika v sintaksi. Po odločitvi PostgreSQL, da želijo iti v smer JSON, so ves trud vložili v JSONB in pustili hstore takšen, kot je sedaj (povzeto iz [81]). Shranjen je v binarnem formatu ter ponuja množico operatorjev in funkcij, ki jih lahko uporabljamo.

Hstore omogoča večjo ali pa vsaj primerljivo hitrost v primerjavi z JSONB [77]. Kot razloga za razliko v hitrosti lahko navedemo dve lastnosti. Podatkovni tip JSONB je gnezdena struktura, medtem ko hstore deluje samo na eni ravni (ključ-vrednost). Poleg tega JSONB podpira vse podatkovne tipe JSON (boolean, array in druge), medtem ko sta oba, ključ in vrednost, pri hstore navadna niza (string).

3.3 Arhitektura za reševanje problema horizontalnega skaliranja

Problematiko horizontalnega skaliranja relacijskih baz smo opisali v razdelku 2.5.4. Navedli smo, da je problematično izvajati stike (operacija JOIN), če so tabele na različnih strežnikih, hkrati pa je v porazdeljeni arhitekturi težko zagotoviti ACID transakcije in sočasno visoko razpoložljivost. V tem razdelku pregledamo različne arhitekture, ki poskušajo rešiti problem horizontalnega skaliranja za relacijske sisteme.

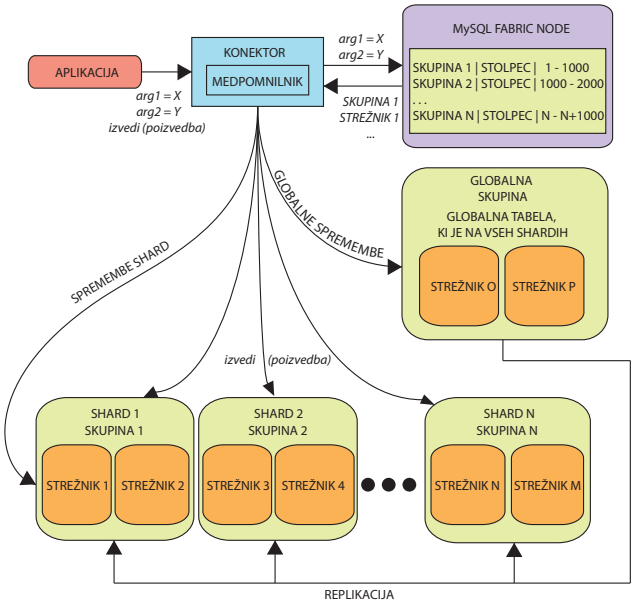
3.3.1 Shranjevalni mehanizem NDB

Shranjevalni mehanizem NDB je mehanizem, na katerem temelji MySQL Cluster. Za razliko od InnoDB je NDB zasnovan za delo z gručo in uvaja sledeče komponente: vozlišča SQL (omogočajo dostop do podatkov), podatkovna vozlišča (hranijo podatke) in upravljalna vozlišča (upravljajo z vozlišči, vsebujejo konfiguracijske podatke, izvajajo varnostne kopije) [82].

3.3.2 MySQL Fabric

MySQL rešuje problem skaliranja in obremenitev z izdelkoma MySQL Cluster in MySQL Fabric. Prvi je zrela in testirana rešitev. Omogoča [83], [84], [62]:

- sinhrono replikacijo;
- hitro reševanje vozlišč v primeru nesreč;
- transparentno deljenje podatkov (deljenje podatkov je doseženo na ravni baze in ne aplikacije);
- visoko razpoložljivost (mehanizem heartbeat za odkrivanje odpovedanih vozlišč, nedeljena arhitektura preprečuje izpad celotne gruče, pokvarjena vozlišča se samodejno ponovno zaženejo in sinhronizirajo);
- NoSQL (Memcached) in SQL način dostopa do podatkov (API za PHP, PERL, Python). Odjemalci lahko pošiljajo poizvedbe SQL in dobijo odgovore iz vozlišč SQL (razdelek 3.3.1), lahko pa uporabijo vmesnik NDB



Slika 3.12: Za delovanje MySQL Fabric je ključno vozlišče MySQL Fabric Node, ki vsebuje informacije, kje se nahajajo podatki.

(vsebuje razrede in funkcije napisane v programskem jeziku C++), s katerim lahko neposredno dostopajo do shranjevalnega mehanizma NDB. Več o tem v viru [85];

- hanjenje vseh podatkov v pomnilniku (dodana je tudi možnost za shranjevanje na disk).

Toda za uporabo MySQL Cluster je treba zamenjati shranjevalni mehanizem InnoDB z NDB, kar pa za določene aplikacije ni lahko. Nekatere lastnosti manjkajo, sintaksa SQL je deloma nezdružjiva, treba je prilagoditi obstoječe tabele in odpraviti ostale pomanjkljivosti. Več o omejitvah je opisano v viru [86].

Da bi uporabniki še vedno lahko uporabljali InnoDB in njegove prednosti, so razvili MySQL Fabric.

MySQL Fabric [87], [88] je ogrodje, ki omogoča skaliranje MySQL. Implementacija MySQL Fabric omogoča deljenje podatkov. MySQL deluje po načinu gospodar-suženj. Imamo vozlišče, ki koordinira vse povezave (MySQL Fabric Node) in če le-ta odpove, sistem ne zna usmerjati zahtev do ustreznih vozlišč.

Deljenje podatkov je izvedeno na sledeči način. Množico strežnikov združujemo v skupine (strežnik 1 in strežnik 2, strežnik 3 in strežnik 4, ...). Strežniki v skupini imajo isto strukturo. V skupini določimo gospodarja, ki prejema posodobitve in jih posreduje ostalim. Vsaka skupina hrani drobec podatkov, t.j. množico vrstic podatkov (tabele, ki imajo po deljenem stolpcu vrednosti na intervalu od-do). Tabele, ki ne vsebujejo deljenega stolpca, so prisotne v vseh drobcih. Za njih moramo definirati globalno skupino, ki bo skrbela, da bodo vse posodobitve na nedeljeni stolpec posredovane vsem skupinam.

MySQL Fabric Node je vozlišče, v katerem definiramo tabelo in stolpec, po katerem bomo delili podatke, tip deljenja podatkov in za vsako zgrajeno skupino definiramo območje vrednosti, ki jih bo hranila.

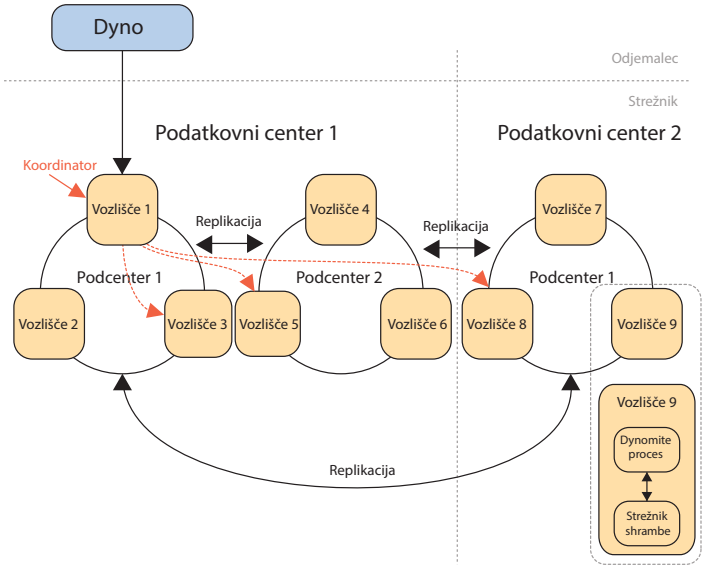
Delovanje strukture je enostavno. Aplikacija se s poizvedbo poveže s konektorjem na MySQL Fabric Node. Ta vrne skupino (ugotovi primerne strežnike skupine), ki vsebuje iskane podatke in stanje vozlišč. Če strežniki niso dovolj zmogljivi, jih lahko prestavimo v drugo skupino. Če nek del postane prezaseden, lahko ustvarimo novo skupino strežnikov in jo dodamo v strukturo.

Struktura v MySQL Fabric je ranljiva, saj je, če odpove MySQL Fabric Node, koordinacija nemogoča.

3.3.3 Dynomite

Podjetje Netflix [31] razvija rešitev, ki bi rešila probleme deljenja podatkov za baze, ki niso zasnovane za porazdeljevanje podatkov. Prvotni cilj je zagotoviti hiter odzivni čas in visoko razpoložljivost. Njihov namen je spremeniti shrambe, ki niso zasnovane za porazdeljevanje, v porazdeljen linearno skalabilen sistem z ohranjanjem protokolov, ki se uporabljajo za te shrambe (npr. Redisov protokol pri komunikaciji z Redis) [31].

Netflix vidi rešitev porazdeljevanja v strukturi, ki ne temelji na strukturi gospodar-suženj (MySQL Fabric). Gručo naj bi sestavljali podatkovni centri, ki se delijo na podcentre (rack), le-te pa vsebujejo vozlišča (slika 3.13). Vsak podcenter vsebuje množico podatkov , ki jo porazdeli na vozlišča. Vsako vozlišče označuje žeton, ki opisuje, kateri del podatkov vsebuje vozlišče. Vozlišče je sestavljeno iz procesa in strežnika shrambe, s katerim proces izmenjuje informacije. Strežniki shrambe so lahko NoSQL sistemi, kot npr. Redis in Memcache (trenutno imple-



Slika 3.13: Dynomite arhitektura omogoča, da odjemalec dostopi do katerega koli vozlišča, le-ta pa ga usmeri do ustreznega vozlišča s podatki.

mentirano) ali pa relacijski sistemi, npr. MySQL (ni še implementirano). Vozlišče se obnaša kot posrednik, koordinator, komunikator in nadzornik povezav.

Pri replikaciji se odjemalec poveže na katero koli vozlišče in želi izvesti zapisovanje. Če podatki spadajo v to vozlišče, kar je razvidno iz žetona, se spremembe zapišejo na lokalni strežnik in nato asinhrono replicirajo na ustrezna vozlišča v ostalih podatkovnih podcentrih in centrih. Če podatki ne spadajo v to vozlišče, vozlišče operacijo preusmeri na ustrezno vozlišče znotraj podatkovnega centra. Ta operacija se hkrati replicira na ustrezna vozlišča v ostalih podcentrih in centrih.

Rešitev, ki jo predstavlja Netflix, se imenuje Dynomite [31]. Zasnova Dynomite zagotavlja visoko razpoložljivost, saj se odjemalec lahko poveže na katero koli vozlišče, če želi brati podatke. Če vozlišče vsebuje podatke, jih posreduje, v nasprotnem primeru pa posreduje zahtevo do ustreznega vozlišča. V primeru odpovedi vozlišča se zahteva pošlje repliciranemu vozlišču v drugem podcentru ali centru. Vozlišče zna vzdrževati članstvo v gruči, odkriti odpoved vozlišča in ga obnoviti po nesreči.

Dynomite lahko uporabljamo z odjemalskimi vmesniki Jedis, Redisson in Spy-Memcached. Toda pri uporabi teh vmesnikov izgubimo lastnosti, ki so implemen-

tirane z odjemalcem Dyno. Dyno je implementacija protokola za komunikacijo s podatkovno shrambo. Razvit je nad popularnimi odjemalci SpyMemcached, Jedis in Redisson z namenom olajšanja migracije. Dyno omogoča ponovno uporabo povezave, preprečuje skok na vozlišče, ki ni lastnik zahtevanih podatkov, nadzira stanja povezav, preusmerja promet od vozlišč, ki morajo biti ugasnjena za potrebe vzdrževanja, preusmerja zahteve po branju na oddaljena vozlišča, če je trenutno vozlišče odpovedalo, in podobno. Zaradi skladnosti z ostalimi omenjenimi odjemalci Dyno omogoča dostop do shramb neposredno z obidenjem Dynomite.

Dynomite ni končan, saj manjkajo nekatere stvari, kot so podpora ostalim SUPB, kriptirana izmenjava informacij med podatkovnimi centri in druge. Ideja je zanimiva in obetajoča, toda trenutno podprta sistema sta nerelacijska sistema tipa ključ-vrednost. Zanimivo bo videti, kako bo Dynomite upravljal z relacijskim sistemom, kot je MySQL. Eden izmed zanimivih in težkih izzivov bo implementacija povezave SQL z Dyno.

3.4 Komercialni sistemi

Na področju prenašanja konceptov so bili aktivni tudi komercialni sistemi. Oracle je v SUPB 12c omogočil, da so lahko podatki sočasno shranjeni v vrstični obliki (in-memory row format), za namene OLTP, in v stolpični obliki (in-memory column format), za analitične odelave podatkov [89]. Oracle 12c podpira tudi JSON, ki se shrani v tip VARCHAR2, CLOB in BLOB. Obstaja množica funkcij za delo z JSON (npr. preverjanje pravilnosti dokumenta JSON), možno je poizvedovati s SQL in ustvariti indekse [90]. Poleg Oracla tudi IBM v DB2 dodaja podporo NoSQL. Dodali so podporo RDF, XML in JSON [91].

3.4.1 Implementacija skaliranja v SQL Server

SQL Server rešuje problem skaliranja z različnimi arhitekurami [92].

1. **Skalabilne deljene podatkovne baze (Scalable Shared Databases)** - Arhitektura temelji na centralizirani bazi, ki je na voljo samo za branje. Nanjo se lahko priključi do osem strežnikov SQL Server. Arhitektura ni primerna, ko imamo opravka z veliko posodabljanji podatkov. Trenutna

arhitektura podpira osem strežnikov in en osrednji strežnik.

2. **Replikacija tipa vsak z vsakim (Peer-To-Peer Replication)** - Vsi strežniki so povezani med sabo in vsak strežnik hrani svojo kopijo podatkov. Uporablja se replikacija za posodobitev vseh kopij. Arhitektura ni primerna, ko veliko strežnikov spreminja podatke, ki se morajo nato posodobiti na vseh kopijah. Ni zgornje meje števila strežnikov, je pa priporočljivo uporabiti do deset strežnikov [93].

3. **Povezani strežniki (Linked servers)** - Arhitektura, v kateri delimo podatke na različne strežnike tako, da je med njimi čim manj sklopljenosti. Potrebno je skrbno načrtovanje sheme. V primeru, da več aplikacij pogosto uporablja dve tabeli, le-teh ni smiselno deliti v dve ločeni bazi. Smiselno pa je deliti tabele, ko neke aplikacije dostopajo samo do ene tabele in druge aplikacije redko še do druge tabele.

4. **Porazdeljeni deljeni pogledi (Distributed Partitioned Views)** - Arhitektura, namenjena transparentnemu skaliranju particioniranih podatkov. Gre za horizontalno skaliranje podatkov glede na vrednost particijskega ključa. Izbrati je treba prave particijske ključe, ki enakomerno porazdelijo podatke in s tem zagotovijo, da se večina sprememb zgodi na enem strežniku. Kljub temu, da naj bi arhitektura omogočala transparentno skaliranje, pa je težko zagotoviti particijsko shemo, ki bi ustrezala vsem aplikacijam, ne da bi bilo treba spremeniti same aplikacije.

5. **Usmerjanje glede na odvisnosti med podatki (Data-Dependent Routing)** - Podobno kot porazdeljeni deljeni pogledi, le da aplikacija ali pa neka vmesna storitev usmerja zahtevo na ustrezno podatkovno bazo. Ideja je, da vse podatke, ki se tičejo točno določene entitete (npr. stranka), shranimo v eni bazi. Ta entiteta je enolično določena s ključem. Vsaka podatkovna baza upravlja poizvedbe nad svojimi entitetami. Od aplikacije je nato odvisno, kako identificira indentifikator (npr. id stranke), ki jo bo povezal z entiteto (npr. prijavno okno v aplikaciji, ki poveže uporabnika z identifikatorjem). Po prejemu identifikatorja potrebujemo mehanizem, ki najde strežnik, na katerem se nanaša ustrezna entiteta.

Poglavje 4

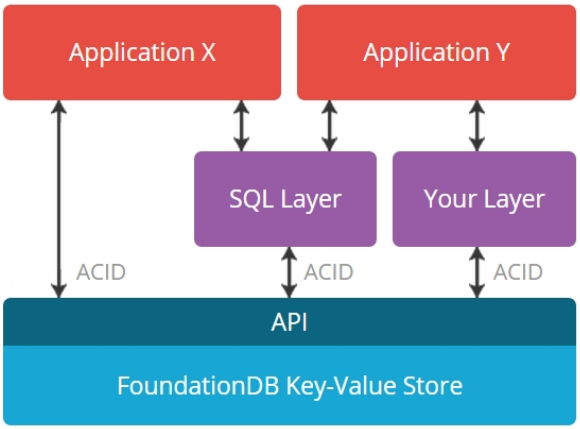
Prenašanje relacijskih konceptov v nerelacijske sisteme

4.1 Podatkovni model in SQL

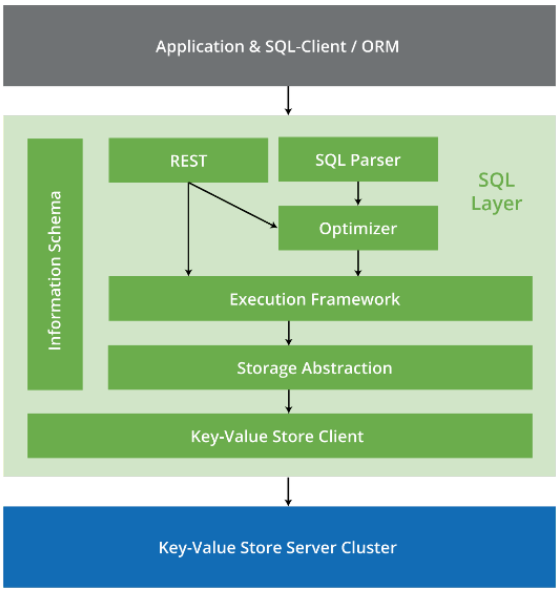
Podjetje FoundationDB [94] je novinec na področju ponudnikov SUPB. Leta 2013 so izdali prvi SUPB, zadnja stabilna različica je bila izdana decembra 2014. Njihova baza je dokaz, da je moč združiti relacijske in nerelacijske koncepte.

FoundationDB je sistem NoSQL tipa urejen ključ-vrednost (sorted key-value), ki zagotavlja ACID transakcije, skalabilnost in odpornost v primeru napake ali izpada (fault tolerance). Vpeljuje koncept plasti v arhitekturo. Arhitekturo sestavljajo trije glavni elementi: aplikacija, plasti in shramba tipa ključ-vrednost.

Na najvišji ravni je aplikacija, ki pošilja podatke plasti ali pa jih posreduje neposredno v shrambo tipa ključ-vrednost. V srednji ravni so plasti, katerih namen je, da implementirajo podatkovni model. FoundationDB loči podatkovni model od načina shranjevanja podatkov. Ideja je, da bi plasti implementirale različne shrambe (relacijske, dokumentne, za grafe) in podatke preko vmesnikov shranile v shrambo tipa ključ-vrednost na najnižji ravni. Trenutno je razvita le plast SQL.



Slika 4.1: Aplikacija lahko komuncira preko plasti SQL ali pa neposredno s shrambo tipa ključ-vrednost. Vir: [95].



Slika 4.2: Koraki, skozi katere gre poizvedba ali ukaz v plasti SQL, da se na koncu pretvori v ustrezne operacije ključ-vrednost. Vir: [94].

4.1.1 Plast SQL

Plast SQL je v celoti ločena od najnižje ležeče shrambe tipa ključ-vrednost. Odjemalec (aplikacija) izmenjuje informacije s plastjo SQL preko SQL, le-ta pa s shrambo tipa ključ-vrednost preko API. Poizvedbe se razčlenijo, pretvorijo in optimizirajo. Zahteve se nato posredujejo abstrakcijski plasti, ki prenese podatke v shrambo ali iz shrambe ključ-vrednost. Plast SQL je združljiva s standardom SQL-92, z omejitvami (2.6), ter omogoča dostop in upravljanje preko ORM (Object Relational Mapping - npr. dostop iz Django), REST API ali neposredno preko ukazne vrstice v plasti SQL.

Plast SQL je brez stanja (stateless), kar pomeni, da za vso stanje skrbi shramba tipa ključ-vrednost [96]. Shramba ključ-vrednost skrbi za trajnost podatkov, replikacijo, deljenje podakov, ACID transakcije in ostalo [96]. Ker plast SQL ne upravlja s stanjem, lahko naenkrat teče več instanc plasti SQL na isti bazi. Plasti SQL se tako lahko poljubno dodaja in odstranjuje, s tem se poveča skalabilnost, saj lahko odjemalec komunicira s poljubno plastjo za vsako ACID transakcijo.

4.1.2 Preslikovanje med SQL in ključ-vrednost

Plast SQL upravlja z informacijami o shemi (tabele in indeksi) in metapodatki. Ključ je kazalec na vrstico v tabeli in vrednost je ta vrstica. Oblika in vrednost ključa sta odvisni od konstrukta, imenovanega skupina tabel (Table-Group), ki predstavlja skupek ene ali več tabel. Za vsako skupino tabel se ustvari mapa z unikatnim bitnim ključem in metapodatki, ki omogočajo poizvedbo po imenu mape. Pri shranjevanju podatkov imamo na voljo tri formate. Pri uporabi formata terka (tuple) se vrstica shrani v format tipa ključ-vrednost. Ključ je skupek predpone imena mape, položaja tabele znotraj skupine tabel in primarnega ključa, vrednost pa predstavlja terka vseh stolpcev v vrstici [94], [97].

Z izvedbo ukaza se ustvari mapa, ki je dostopna preko vrednosti ali imena (metapodatki).

Ker vemo, kako se podatki iz SQL preslikajo v obliko ključ-vrednost, lahko podatke beremo neposredno iz shrambe s poizvedbo tipa ključ-vrednost.

CREATE TABLE shema1.tabela1(id INT PRIMARY KEY, a CHAR(10));

mapa	ključ mape	terka, ki opisuje mapo
sql/data/table/shema1/tabela1	\x15\x64	(100)

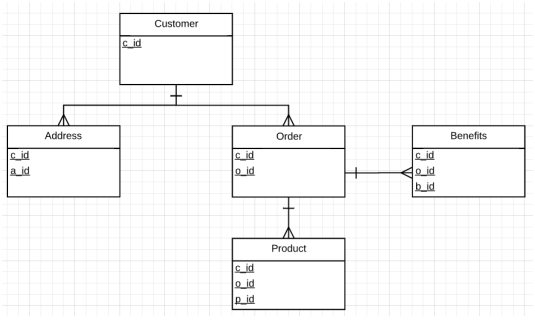
INSERT INTO shema1.tabela1 VALUES (1, 'lep'), (2, 'dan')

ključ	vrednost	ključ v obliki terka	vrednost v formatu terka
\x15\x64\x15\x01\x15\x01	\x15\x01\x01lep\x00	(100,1,1)	(1,'lep')
\x15\x64\x15\x01\x15\x02	\x15\x01\x02dan\x00	(100,1,2)	(2, 'dan')

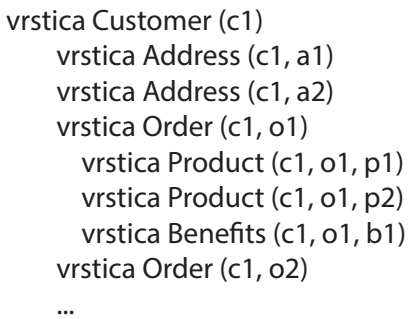
Slika 4.3: Stavki SQL se pretvorijo v ustrezne ključ-vrednost operacije, ki podatke shranijo v obliki ključ-vrednost. Podatki obarvani modro pomenijo, da je tabela1 prva v hierarhiji tabel.

4.1.3 Skupina tabel

Skupina tabel je hierarhična skupina povezanih tabel. Skupine tabel definirajo način, preko katerega so fizični podatki shranjeni in vplivajo na način delovanja plasti SQL. Tabele in podatki v skupini so predstavljeni v hierarhiji vrstic, ki opisujejo strukturo. Podatki različnih tabel, ki so med seboj povezani, se prepletajo v isti mapi.



Slika 4.4: Hierarhija tabel (Skupina tabel). V hierarhiji so možne le povezave ena proti več.



Slika 4.5: Shema plasti SQL je urejena v hierarhijo. Sorodne vrstice so med seboj prepletene.

V hierarhiji s slike 4.4 vsaki vrstici Customer sledi vrstica za Address ali vrstica Order (vrstni red ni pomemben, važno je, da se ohranja hierarhija). Vsaki vrstici Order sledi vrstica Product ali Benefits in tako dalje.

Prepletanje vrstic prinaša prednosti, saj zagotavlja zaporeden dostop za večino branj in pisanj. Podatki o entiteti so shranjeni skupaj (slika 4.5), kar omogoča večjo manipulacijo z uporabo vgnezdenih poizvedb SQL. Skupina tabel zamenjuje tabele kot osnovno fizično enoto v plasti SQL. Poleg hierarhije upravlja tudi s hrambenimi strukturami, vodenjem poizvedb skozi optimizacijske procese in drugim.
Obstajajo sledeča pravila pri sestavljanju tabel:

- Skupina tabel je sestavljena iz ene ali več tabel.
- Skupina tabel je v hierarhični obliki z enim staršem in povezavo tipa ena proti mnogo med starševskimi tabelami in vsemi otroki.
- Vsaka tabela je v natančno eni skupini, četudi je v njej sama.
- Tabele v relaciji mnogo proti mnogo morajo obstajati v le eni skupini.

4.1.4 Izboljšave v zadnji različici

FoundationDB z različico 1 ni dosegel zastavljenih ciljev za nizko latenco in skalabilnost (400 000 operacij zapisovanja na sekundo napram cilju 10 000 000). Razlog je bil, da je transakcijski mehanizem zgrajen v centralizirani obliki z enim glavnim

strojem, ki je predstavljal ozko grlo. Omejitve so opisane v [98]. V različici 3, ki je bila izdana 10. decembra 2014, so prenovili transakcijski mehanizem in prepisali tri pomembne komponente mehanizma: posrednike (proxies), reševalce izolacije (resolver) in transakcijske zabeležke [99]. Njihove naloge so opisane skozi sledeči primer.

1. Odjemalec pošlje transakcijo na posrednika, ki skladišči prihajajoče transakcije in jih upravlja v vrstah.

2. Posrednik v drugem koraku preko reševalcev izolacije preveri, ali bo transakcija izolirana. Reševalci izolacije beležijo zgodovino sprememb ključev in preverjajo, ali so transakcije izolirane.

3. V tretjem koraku posredniki pošljejo transakcije na transakcijske zabeležke, če so transakcije prestale izolacijska preverjanja. Transakcijske zabeležke shranijo transakcije in zabeležke na disk ter posredujejo odgovor odjemalcu.

4. V zadnjem koraku transakcijske zabeležke trajno zapišejo spremembe na vozlišča, ki so namenjena hrambi podatkov.

S prenovo transakcijskega mehanizma naj bi po testu v [99] dosegli 14 400 000 naključnih zapisovanj na sekundo, kar za faktor 14 prekaša rezultate, ki jih je Netflix dosegel s sistemom Cassandra [99], poleg tega pa še ohranja ACID transakcije.

Poglavje 5

Novi relacijski sistemi (NewSQL)

Podatkovne baze NewSQL so relacijske podatkovne baze, ki so zasnovane z namenom podpiranja ACID transakcij, obdelave transakcij (OLTP) v realnem času in analitične obdelave velikih količin podatkov (OLAP) [13, stran 1]. To dosegajo z uporabo NoSQL značilnosti, kot so vzporedno procesiranje, obdelava podatkov znotraj pomnilnika (in-memory processing), uporaba stolpičnih shramb in podobno [13, stran 1, 2]. Na NewSQL lahko gledamo kot na nekakšen poskus združenja relacijskih in nerelacijskih sistemov. Primeri nekaterih popularnih NewSQL so Clustrix [100], NuoDB [101], VoltDB [102], MemSQL [103] in drugi.

5.1 Sistem VoltDB

VoltDB [104] je NewSQL SUPB, zasnovan po ugotovitvah članka [35]. Za razliko od relacijskih SUPB se vsa obdelava podatkov dogaja v pomnilniku in tako odpravlja potrebo po upravljalcu medpomnilnika. Temelji na avtomatičnem particioniranju podatkov, kar poveča prepustnost in s tem omogoča vzporedno izvajanje več ACID transakcij. VoltDB in njegove particije so enonitne (single threaded) in med seboj neodvisne, kar odpravi potrebo po zaklepanju in upravljanju deljenih struktur. Podatki in procesiranje nad podatki je particionirano skupaj in porazdeljeno čez več procesorskih jeder v nedeljeni arhitekturi (shared nothing arhitecture) [102].

Je porazdeljen sistem, ki omogoča skoraj linearno horizontalno skaliranje.

5.1.1 Vzpostavitev podatkovne baze

Pri VoltDB je treba definirati shemo pred ustvarjanjem podatkovne baze. S tem lahko VoltDB določi najboljši možen način razvrstitve particij, ko bazo zaženemo. V prvem je treba ustvariti shemo, poglede in indekse preko standardnih stavkov SQL (CREATE TABLE, CREATE INDEX, CREATE VIEW), dodati definicijo shranjenih procedur (stavki SQL, razredi v Javi) in dodati definicijo particijskih tabel.

Vse našteto se zapiše v **aplikacijski katalog**, katerega kopijo dobi vsako vozlišče. Katalog se v nadaljevanju prevede in preko njega se zažene podatkovna baza. Pri vzpostavljanju gruče je treba podati še t.i. izvozno datoteko (deployment file), ki opisuje gručo, particioniranje in ostale podatke.

5.1.2 Shranjene procedure

Shranjene procedure so transakcijska enota. Prevedejo se pred zagonom baze (prevajanje aplikacijskega kataloga), zato so načrtovane vnaprej in so optimalno nastavljene. Lahko so enostavne, npr. stavek

SELECT CITY FROM CITIES WHERE COUNTRY=?,

ali pa kompleksne. Kompleksne lahko vsebujejo več poizvedb, pogojne vejitve in ostalo logiko. Za kompleksne procedure je treba napisati Java razred.

5.1.3 Particioniranje

Particioniranje je najpomembnejši del sistema VoltDB, saj se particionira podatke (tabele) in delo (shranjene procedure). Pravilno particioniranje v sistemu VoltDB omogoča doseči najboljše rezultate, a v ta namen moramo particijske tabele definirati sami na način, ki bo sistemu VoltDB omogočil, da bo lahko izvedel več enoparticijskih poizvedb vzporedno.

Denimo, da imamo tabelo s strukturo: igralec (igralecId, ime, priimek). Tabelo particioniramo tako, da definiramo particijski stolpec (PARTITION TABLE igralec ON COLUMN igralecId). Podatki se pošiljajo na različne particije glede na

vrednost particijskega stolpca (poizvedbe se lahko izvajajo vzporedno na različnih particijah). VoltDB najboljše rezultate dosega, kadar poizvedba obišče le eno particijo (enoparticijske poizvedbe). Ne gredo pa vse poizvedbe skozi samo eno particijo.

Če v poizvedbi iščemo po napačnem stolpcu (v našem primeru imamo particijski stolec igralecId), bi poizvedba

SELECT * FROM igralec WHERE priimek = "Novak"

potekala skozi vse particije, medtem ko bi poizvedba

SELECT * FROM igralec WHERE igralecId=32

potekala po eni sami particiji.

Poleg podatkov se particionirajo tudi shranjene procedure z definiranjem particijske tabele in stolpca (PARTITION PROCEDURE dobiIgralca ON TABLE igralec COLUMN IgralecId). Shranjene procedure so usmerjene k primerni particiji glede na vrednost posredovanega argumenta. Paziti je treba, da so vse poizvedbe znotraj kompleksnih procedur pravilno particionirane. To pomeni, da nobena ne išče po parametru, ki ni particioniran (npr. priimek), saj to ne bo učinkovito.

5.1.4 Replikacija

Priporočljivo je particionirati vsako tabelo. Če tabela ni particionirana, se ta replicira na vse particije. **Prednost** replikacije je, da omogoča združevanje particioniranih in repliciranih tabel in s tem dobimo več poizvedb enoparticijskih. Ker so podatki na voljo v več particijah, se izboljša razpoložljivost. **Slabost** pristopa je, da so stavki INSERT in UPDATE večparticijske transakcije. Zato je najboljša praksa replicirati majhne tabele, ki se ne spreminjajo (read-only) in se uporabljajo za t.i. lookup poizvedbe. Replikacija particij je znana pod imenom K-Safety [105] in pove, koliko izgub vozlišč lahko baza prenese.

5.1.5 Trajnost in porazdeljene ACID transakcije

Za trajnost je poskrbljeno preko treh storitev.

- SNAPSHOT je posnetek baze v nekem času, ki se uporabi za kopijo.

- Beleženje ukazov beleži transakcije od zadnjega posnetka baze. Pri vzpostavitvi baze po nesreči se najprej uporabi posnetek baze, nato pa se zaporedno izvaja vse transakcije, zabeležene pri beleženju.
- Replikacija podatkovne baze na drugo gručo.

Iz vidika performans naj bi bil VoltDB za faktor 100 hitrejši od MySQL in za faktor 14 hitrejši od sistema Cassandra [106]. A izkaže se, da ima VoltDB omejitve pri izvajanju porazdeljenih ACID transakcij. Za zagotavljanje le-teh uporablja t.i. two-phased commit protocol, ki zaklene celotno podatkovno bazo, povzroči veliko omrežnih zakasnitev (zaglavja) in s tem omejuje količino opravljenega dela (od nekaj sto do nekaj tisoč transakcij na sekundo) [107].

5.1.6 Optimizacija poizvedb

VoltDB ustvari izvajalni načrt za vsako poizvedbo v vsaki proceduri in načrte shrani v podmapo in aplikacijski katalog [108]. VoltDB ne ve, koliko zapisov bo imela tabela v času prevedbe in pri operacijah stika tabele morda ne bodo optimalno združene [109]. Izvajalni načrt je moč spremeniti pri definiranju poizvedbe, tako da podamo zaporedje tabel pri stiku (zadnja vrstica v spodnjem primeru). Če izvedemo ad-hoc poizvedbo znotraj baze, ni nujno, da se bo izvedla po istem načrtu kot shranjena procedura.

```
SELECT ...
FROM ...
WHERE ...
ORDER BY ...
" manager , department , employee "
```

Slika 5.1: Poizvedbo lahko optimiziramo tako, da v zadnji vrstici navedemo vrstni red tabel, po katerem bodo slednje združene.

5.1.7 Gruče

Gruča se vzpostavi tako, da se zaženejo vsa vozlišča hkrati, pri tem pa se določi vozlišče, ki bo koordiniralo vzpostavljanje gruče (host). Nastavitve gruče se določa preko izvozne datoteke. V tej je določeno vozlišče, ki bo koordinator pri vzpostavljanju gruče (host), število vozlišč v gruči, število particij, ki se ustvarijo na vsakemu vozlišču (siteperhost), vrednost K-Safety (kfactor) in po potrebi ostali parametri.

5.1.8 Opombe

VoltDB uporablja omejeno različico standarda SQL-99 [106]. Potreben je natančen pregled sintakse za ustvarjanje tabele in nastavitev indeksov, particijskih tabel in omejitve NOT NULL ter ASSUMEUNIQUE. VoltDB ne podpira tujih ključev (foreign keys), omejitev (check) in sprožilcev, stavkov ALTER in DROP. Upravljanje z uporabniki, skupinami in pravicami poteka preko datotek XML. Vsak stolpec je lahko velik največ 1 MB, vsi stolpci tabele ne smejo preseči 2 MB in particijski stolpec ne more biti null. VoltDB lahko zagotovi unikatnost (uniqueness), če indeks vsebuje particijski stolpec. Unikatnost ne more biti zagotovljena skozi celotno bazo, ampak samo čez particijo. Če želimo zagotoviti unikaten stolpec, ki ni del particijskega stolpca, moramo uporabiti besedo ASSUMEUNIQUE, ki zagotavlja edinstvenost na trenutni particiji. Razvijalci morajo poskrbeti, da bo unikatnost tudi globalna [110]. Dve ali več tabel lahko združimo, če so particionirane po isti vrednosti in po particijskem stolpcu z enakostjo (=). Združevanje dveh tabel po neparticijskem stolpcu ali uporabi drugega operatorja (različen od =) ni podprto. Ni pa omejitev pri združevanju repliciranih tabel [111].

5.1.9 Povzetek

VoltDB izvaja operacije v pomnilniku. Moč VoltDB izvira iz particioniranja. Pravilna izbira particijskega stolpca in pravilno zastavljene procedure omogočajo najboljše rezultate. VoltDB razpoložljivost veča z replikacijo, pri kateri je optimalno replicirati majhne tabele, ki se ne posodabljajo pogosto.

Pri vzpostavljanju gruče je treba uporabiti izvozno datoteko, ki opredeljuje samo gručo, in določiti koordinatorja. Trajnost podatkov je zagotovljena preko

posnetkov baze, beleženja ukazov in replikacije podatkovne baze. Z VoltDB lahko upravljamo preko ukazne vrstice in ostalih vmesnikov za interaktivne poizvedbe (JDBC in JSON) ter odjemalskih vmesnikov za Java, C++, C#, PHP, Python, Node.js, Erlang, Go in Ruby. VoltDB podpira okrnjeno podmnožico standarda SQL-99. Ni mogoče spreminjati in brisati tabel med delovanjem baze, upravljanje z uporabniki poteka preko XML datotek, za globalno unikatnost podatkov na vseh tabelah ("unique" lastnost na stolpcih, ki niso del particioniranega stolpca) morajo skrbeti razvijalci in še druge pomanjkljivosti.

Poglavje 6

Ocenjevanje prenesenih konceptov

6.1 Primerjalni testi

Namen tega poglavja je prikazati uspešnost prenesenih konceptov iz prejšnjega poglavja s primerjalnimi testi (benchmark). S primerjalnimi testi se opredeli uspešnost in kakovost delovanja podatkovnih baz, zato ponudniki teste navadno prikrojijo ali pa uporabijo trike [112] (pomnenje vseh vrstic v pomnilnik, shranjevanje izvedbenih načrtov SQL v pomnilnik, uporaba specializiranih strojev, arhitektur in drugih trikov), da pokažejo svoj sistem za najboljšega na svojem področju.

Da bi zagotovili neko mero pravičnosti primerjav podatkovnih baz, je bil ustanovljen **Transaction Processing Performance Council (TPC) [113]**. Namen TPC je definirati množice funkcijskih zahtev, ki jih lahko testiramo na transakcijskih sistemih neodvisno od strojne opreme ali operacijskega sistema [113]. Ponudniki lahko nato izdelajo primerjalne teste TPC in, če le ti ustrezajo vsem funkcijskim zahtevam, zagotovijo končnim uporabnikom pravično medsebojno primerjavo baz [113]. Da so rezultati uradno sprejeti, mora postopek testiranja iti skozi revizijo in biti potrjen s strani TPC. Obstaja več kategorij primerjalnih testov TPC in navadno jih uporabljajo relacijski SUPB.

Obstaja veliko različnih orodij, ki izvajajo primerjalne teste (TPC-C) za rela-

cijske sisteme (BenchmarkSQL [114], DBT2 [115], HammerDB [116], TPCC-UVa [117] in ostali). Podrobneje smo si pogledali BenchmarkSQL, ki vsebuje Java implementacijo TPC-C primerjalnega testa, imenovano jTPCC.

Na drugi strani sistemi NoSQL nimajo standardnih primerjalnih testov [118]. Teste je težko pisati, saj se sistemi razlikujejo po načinu hranjenja in obdelave podatkov [118] ter so lahko učinkoviti v določenih primerih uporabe in ne v drugih. Poleg tega se sistemi za Big Data nenehno razvijajo, spreminjajo se podatki, sheme in načini manipulacije podatkov, kar oteži pisanje primerjalnih testov, saj morajo ti slediti spremembam. Porazdeljena zasnova NoSQL z različnimi viri in oblikami podatkov oteži testiranje skaliranja, saj skaliranje poteka na več načinov in je težko zagotoviti, da bo problem primerjalnega testa pravilno skaliran [118]. Poleg tega večplastna in porazdeljena zasnova NoSQL sistemov oteži razvoj enostavnih testov.

Večina NoSQL sistemov ima lastne primerjalne teste, npr. Redis [119], MongoDB [120]. Toda zasledili smo ([121], [122], [123] in drugi), da se za primerjavo NoSQL sistemov največkrat uporablja verjetno najbolj znano orodje **Yahoo Cloud Serving Benchmark (YCSB)** [124]. YCSB je postal de facto standardno orodje za primerjanje baz v oblaku [125]. Privzeto vsebuje množico obremenitvenih testov, s katerimi testiramo hitrost različnih shramb ključ-vrednost ali oblačnih shramb (HBase, Cassandra, MongoDB in ostale). Poleg privzetih testov omogoča razširjanje orodja z lastnimi obremenitvenimi scenariji in dodajanjem podpore novim podatkovnim bazam.

V splošnem je podatkovne baze med seboj težko testirati objektivno in pravično, saj so si tehnološko zelo različne. Tečejo na različni strojni opremi, operacijskih sistemih in imajo svoje značilnosti, ki jih naredijo posebne. Namen tega poglavja je prikazati **uspešnost prenesenih konceptov**. Za testiranje prenesenih konceptov smo napisali enostavne teste. Standardnih testov nismo uporabili zaradi sledečih razlogov:

- Standardni primerjalni testi so bolj kompleksni, kot jih potrebujemo.
- Standardni primerjalni testi so težki za implementacijo in je za njih potrebno napredno poznavanje podatkovnih baz.
- Standardni testi za NoSQL ne obstajajo.

- Določeni sistemi so optimirani za različne probleme, a imajo pomanjkljivosti. Npr. VoltDB optimizira poizvedbe pred ustvarjanjem podatkovne baze, a ne podpira niti standarda SQL-92, kar pomeni, da določenih lastnosti v primerjavi z relacijskimi sistemi nima. HandlerSocket vtičnik za MySQL in MariaDB omogoča hitro vstavljanje in pridobivanje podatkov, a se odpove validaciji (ni validacije pri vnosu podatkov v stolpce tabel).
- Vsi preneseni koncepti (HandlerSocket, Memcached API, shranjevalni mehanizem Connect in drugi) so svojevrstni in njihova uporaba je raznolika (za specifične probleme), kar oteži pisanje testov.
- Testi morajo iti skozi revizijo in biti potrjeni s strani TPC.

Standardni primerjalni testi nudijo neko zanesljivo primerjavo. V magistrskem delu ne bomo implementirali standardnih primerjalnih testov zaradi zgoraj naštetih razlogov. Namen magistrskega dela ni dokazati, kateri sistemi so hitrejši in boljši, temveč podati oceno, v kolikšni meri so preneseni koncepti implementirani.

6.2 Kvantitativni testi

V kvantitativnih testih smo v večini primerov merili hitrost operacij vstavljanja in poizvedovanja. Testi so napisani v jeziku Java in izvedeni na prenosniku, na katerem smo uporabili Vagrant [126] za vzpostavitev virutalnega okolja. Ustvarili smo sledeče teste za prenesene koncepte:

- **Memcached API:** Hitrost vstavljanja in poizvedovanja, Memcached API proti SQL.
- **HandlerSocket:** Hitrost vstavljanja in poizvedovanja, HandlerSocket proti SQL.
- **Dinamični stolpci:** Hitrost vstavljanja in poizvedovanja, dinamični stolpci proti običajnim tabelam.
- **JSONB:** Hitrost vstavljanja in poizvedovanja, JSONB proti MongoDB.

- **Shranjevalni mehanizem TokuDB:** Primerjava kompresije, hitrosti vstavljanja in spreminjanja ter hitrost dodajanja stolpca in indeksa na obstoječi tabeli, InnoDB (XtraDB) proti TokuDB.
- **Shranjevalni mehanizem Cassandra:** Hitrost vstavljanja, brisanja, spreminjanja in poizvedovanja, MariaDB tabele proti MariaDB tabelami, ki so povezane s s sistemom Cassandra.

6.3 Kvalitativni testi

V kvalitativnih testih smo podali primere uporabe konceptov, omenili morebitne prednosti in slabosti ter opravili primerjave konceptov (sintaksa SQL pri JSONB napram poizvedovanju v MongoDB, sintaksa SQL pri uporabi dinamičnih stolpcev napram sintaksi pri uporabi JSONB in podobno). Poleg zgoraj omenjenih konceptov smo kvalitativno ocenili še shranjevalna mehanizma Connect in OQGraph.

6.4 Memcached API

Kvantitativni test

Testirali smo, kako hitro se preko vtičnika in uporabe SQL vstavi zaporedno 50 000 zapisov in kako hitro se izvede 100 000 zaporednih poizvedb. Uporabili smo več knjižnic, saj smo ugotovili, da se rezultati med knjižnicami zelo razlikujejo. Rezultati so bili presenetljivi. Izkaže se, da pri vstavljanju ni velikih razlik med knjižnicami, pri poizvedovanju pa so razlike velike. Izkaže se, da uporaba znanih Javanskih knjižnic (XMemcached, SpyMemcached) ne da dobrih rezultatov. Uporaba pythonove knjižnice se izkaže za najhitrejšo rešitev pri poizvedovanju. Testi morda ne odražajo dejanskega stanja, saj je možno, da so uporabljene knjižnice neprimerne. Morda se memcached API bolje odreže, ko imamo opravka z več nitmi, kar v tem delu nismo obravnavali.

Opombe

V paru smo uporabili Java-Memcached-Client in SpyMemcached, ker smo naleteli na težave pri prebiranju podatkov z Java-Memcached-Client (težava pri get metodi) in vstavljanju podatkov s SpyMemcached (ne zapiše vseh podatkov in je

gonilniki	vstavljanje	poizvedovanje
SQL (gonilnik Java)	139 s	138 s
MySQLdb + python-memcached-1.53	153 s	73 s
XMemcachedClient	143 s	290 s
Memcached-Java-Client (vstavljanje) SpyMemcached (poizvedovanje)	130 s	289 s

Tabela 6.1: Testiranje učinkovitosti pri uporabi različnih knjižnic.

```
MemcachedClient mcc= new MemcachedClient(new InetSocketAddress("192.168.50.111", 11211));
Random rand = new Random();

long startTime = System.currentTimeMillis();
for(int i=0; i<queries; i++)
{
     try
     {
        mcc.get(rand.nextInt(inserts)+"");
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
long endTime = System.currentTimeMillis();
```

Slika 6.1: Operacija get ima navadno en argument, in sicer ključ, po katerem iščemo.

slabši pod obremenitvijo). Način pridobivanja podatkov iz shrambe je podoben pri vseh knjižnicah preko metode get.

6.5 HandlerSocket

Kvantitativna ocena

Primerjali smo hitrost vstavljanja in poizvedovanja HandlerSocket v primerjavi s SQL. HandlerSocket se bolje odreže kot uporaba SQL pri vstavljanju in preprostem poizvedovanju. Prednost HandlerSocket je skupinska obdelava, ki sicer malo zakasni odgovor, toda omogoča nam obdelavo večje količine podatkov. Uporabili smo knjižnico handlersocketforjava. Rezultate prikazuje tabela 6.2. Zavedati se je treba, da se HandlerSocket v zameno za obidenje plasti SQL odpove določenim lastnostim, kot npr. validaciji vnesenih podatkov, sprožilcem, agregacijam in ostalim

```
for(int i=0; i<=size; i++) {
    try {
        ResultSet rs = st.executeQuery("SELECT * FROM "+table+" WHERE id="+x.nextInt(2000000)+"");
        while(rs.next())
            y.add(rs.getString(1));
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Slika 6.2: Primer SQL poizvedbe, ki je ekvivalent HandlerSocket poizvedbi na sliki 6.3, le da ne vključuje skupinske obdelave.

operacije	SQL	HandlerSocket	št. operacij
vstavljanje	45,2 s	28,2 s	2000000
poizvedovanje (sliki 6.2, 6.3)	49,3 s	1,7 s	100000

Tabela 6.2: Testiranje učinkovitosti HandlerSocket.

lastnostim (omejitve so opisane v 3.2.2.3).

Kvalitativna ocena

Sintaksa HandlerSocket ni samoumevna, za razliko od SQL. Protokol je enostaven, tekstoven. Ukazi, zahteve in odgovori so dolgi eno vrstico, ki mora biti oblikovana po pravilih z določenimi znaki. Obstajajo knjižnice za različne jezike, ki poenostavijo uporabo.

```
/*              HANDLER SOCKET                  */
boolean verbose = true;
String host = "192.168.50.110";
int port = 9998;
String id = "1";
String db = "mariadb_test";
String table ="handlersocket";
String fieldList = "id,name,lastname,age,commune,income,outcome";
String index ="PRIMARY";
int size = 100000;
int loop = 100;
```

```
HandlerSocket hs = new HandlerSocket();
long start = System.currentTimeMillis();
int i=0;
ArrayList<String> y = new ArrayList<String>();

try
{
    hs.open(host, port);
    hs.command().openIndex(id, db, table, index, fieldList);
    /* Odpremo povezavo, ki bo odprta dokler odjemalec ne konča.
     * Podamo podatkovno bazo(db), tabelo (table), ali bomo iskali
     * po primarnem ključu (index), in seznam polj katere bomo ali
     * pridobili ali v njih vstavljali ali v njih spreminjali podatke
     * (fieldList). Parameter id je število, s katerim referenciramo
     * te nastavitve.
     */
    for(i = 0 ; i <= size ; i++){
         if(i != 0 && i % loop == 0){
            /* na vsak 100 korak */
            List<HandlerSocketResult> results = hs.execute();
            /*  izvedemo skupinsko obdelavo 100ih poizvedb   */
            if(verbose){
                for(HandlerSocketResult result : results){
                    /*  Rezultate izpišemo   */
                    y.add(result.toString());
                }
            }
            if(i == size)
                break;
        }
        String[] keys = new String[]{x.nextInt(2000000)+""};
        /* dodamo 'where' pogoj, primarni ključ = x.nextInt(2000000) */
        hs.command().find(id, keys);
        /* Ukaz se pretvori v ustrezen HandlerSoket izraz in se doda
         * v vrsto za izvedbo (skupinska obdelava) BlockingQueue.
         * Pri izvedbi ukaza hs.execute() se izvedejo vsi ukazi v vrsti.
         */
    }
```

Slika 6.3: Primer kode za HandlerSocket. Zgornji primer je enakovreden sledečemu stavku SQL: SELECT id, name, lastname, age, commune, income, outcome FROM mariadb_test.handlersocket WHERE id=?.

6.6 Dinamični stolpci

Kvantitativna ocena

Dinamični stolpci omogočajo več atributov na posamezno vrstico, a njihova trenutna izvedba ne omogoča hitrega izvajanja operacij, saj so časi vstavljanja in poizvedovanja dolgi v primerjavi z vstavljanjem in poizvedovanjem v tabelo brez dinamičnih stolpcev. Rezultati so prikazani v tabeli 6.3. Slika 6.6 vsebuje poizvedbe, ki se nanašajo na sliko 6.5.

Kvalitativna ocena

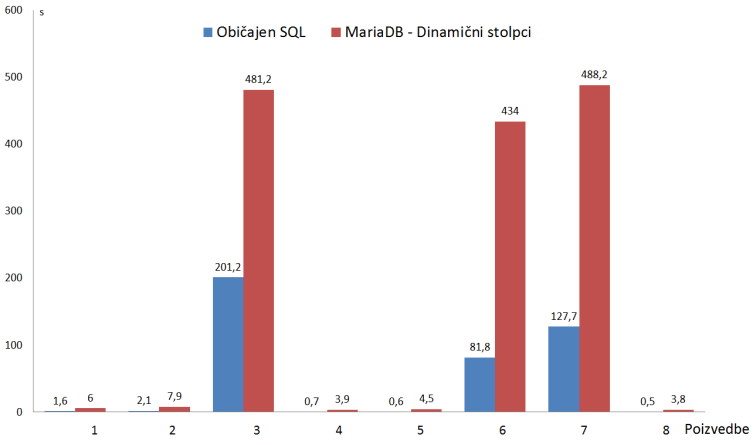
Dinamični stolpci omogočajo prilagajanje sheme, a ta še ni tako izpopolnjena kot JSON. Dodati bi bilo treba še podporo poljem (array) JSON, razčlenjevanju objektov JSON, napredni manipulaciji dokumenta JSON in podpori vsem tipom v JSON [73]. Obstaja način indeksiranja preko virtualnih stolpcev, ki pa še ni najboljši. Čeprav je sintaksa dinamičnih stolpcev enostavna in vgrajena v SQL, je v primerjavi z JSONB od PostgreSQL dolga (predvsem uporaba COLUMN_GET pri gnezdenju) in manj pregledna.

	čas vstavljanja	št. zapisov
dinamični stolpci	153 s	2000000
tabele brez dinamičnih stolpcev	97 s	2000000

Tabela 6.3: Hitrost vstavljanja v dinamične stolpce napram vstavljanju v tabelo brez dinamičnih stolpcev.

```
CREATE TABLE person (data blob);
INSERT INTO person VALUES (COLUMN_CREATE("name", "Janez", "lastname", "Novak", "age",
15, "commune", "Ljubljana", "earnings", COLUMN_CREATE("income", 1000, "outcome", 500)));

CREATE TABLE personsql (id INTEGER, name VARCHAR(20), lastname VARCHAR(30), age INTEGER,
commune VARCHAR(50), income INTEGER, outcome INTEGER, PRIMARY KEY(id)) ENGINE=InnoDB;
```

Slika 6.4: Primer kode, s katero smo zgradili in vstavili podatke v tabelo z dinamičnimi stolpci, in koda za tabelo brez dinamičnih stolpcev.



Slika 6.5: Hitrost poizvedovanja po dinamičnih stolpcih napram tabeli brez dinamičnih stolpcev. Poizvedbe so opisane na sliki 6.6.

```
1.
SELECT name FROM personsql WHERE commune = "Ajdovščina";
SELECT COLUMN_GET(data, 'name' as CHAR) FROM person WHERE
COLUMN_GET(data, 'commune' as CHAR) = 'Ajdovščina';

2.
SELECT distinct commune from personsql;
SELECT distinct COLUMN_GET(data, 'commune' as CHAR) from person;

3.
SELECT name FROM personsql;
SELECT COLUMN_GET(data, 'name' as CHAR) FROM person;

4.
SELECT name  FROM personsql WHERE commune = "Ajdovščina" and age > 50;
SELECT COLUMN_GET(data, 'name' as CHAR)  FROM person WHERE
COLUMN_GET(data, 'commune' as CHAR) = 'Ajdovščina' AND COLUMN_GET(data,
'age' as INTEGER) > 50;

5.
SELECT name  FROM personsql where income = outcome;
SELECT COLUMN_GET(data, 'name' as CHAR)  FROM person where
COLUMN_GET(COLUMN_GET(data, 'earnings' as char), 'income' as INTEGER) =
COLUMN_GET(COLUMN_GET(data, 'earnings' as char), 'outcome' as INTEGER);

6.
SELECT name FROM personsql where age > 50;
SELECT COLUMN_GET(data, 'name' as CHAR)  FROM person where
COLUMN_GET(data, 'age' as INTEGER)>50;

7.
SELECT name  FROM personsql where income > 1500;
SELECT COLUMN_GET(data, 'name' as CHAR)  FROM person where
COLUMN_GET(COLUMN_GET(data, 'earnings' as char), 'income' as INTEGER) > 1500;

8.
SELECT COUNT(*) FROM personsql where lastname = "Novak" and outcome > 2000;
SELECT COUNT(*) FROM person where COLUMN_GET(data, 'lastname' as CHAR) =
'Novak' and  COLUMN_GET(COLUMN_GET(data, 'earnings' as char), 'outcome' as
INTEGER) > 2000;
```

Slika 6.6: Rdeče obarvano besedilo je koda za dinamične stolpce (slika 6.5), modro obarvano pa je koda brez uporabe dinamičnih stolpcev.

```
SELECT COUNT(*) FROM person where
COLUMN_GET(data, 'lastname' as CHAR) = 'Novak' and
COLUMN_GET(COLUMN_GET(data, 'earnings' as char),
                'outcome' as INTEGER) > 2000; ");

SELECT COUNT(*) FROM test where data @>
{"lastname" : "Novak"} and
(data->'earnings'->>'outcome')::int > 2000;
```

Slika 6.7: Primerjava sintakse SQL med dinamičnimi stolpci in JSONB.

6.7 Shranjevalni mehanizem Connect

Kvalitativna ocena

Uporabo shranjevalnega mehanizma Connect je najlažje prikazati na primeru. Prikazali bomo, kako dve datoteki (movies.xml, person.tsv) in tabelo (comment_connect_mysql) na oddaljenemu strežniku preslikamo v MariaDB tabele in jih nato med seboj povezujemo.

Imamo dve datoteki, movies.xml inperson.tsv, ter tabelo comment_connect_mysql (slika 6.8). Ustvariti moramo tri tabele za dve datoteki in eno tabelo MySQL, ki se nanaša na oddaljeno MySQL tabelo. Slika 6.9 prikazuje primer kode za datoteko XML. Sedaj lahko upravljamo s podatki preko SQL (slika 6.10).

Shranjevalni mehanizem Connect poenostavi delo, saj odpravi ročno kodirano programsko kodo. V našem primeru smo odpravili razčlenjevanje XML in tekstovne datoteke s tem, da smo definirali tabele, logika v shranjevalnem mehanizmu Connect pa je poskrbela, da se je vse pravilno preneslo. Odslej lahko kompleksne poizvedbe izvajamo preko SQL. Programiranje logike razčlenjevanja in povezovanja podatkov brez uporabe shranjevalnega mehanizma Connect bi bilo zahtevno, poleg tega pa bi bilo treba spremeniti programsko kodo vsakokrat, ko bi se notranja struktura datotek spremenila.

```
(movies.xml)
<?xml version="1.0" encoding="UTF-8"?>
<movies SUBJECT="XML">
 <movie lang="eng">
  <title>Ong_bak</title>
  <year>2003</year>
  <rating>7</rating>
  <genre>Action,Adventure,Crime,Thriller</genre>
  <cast>
   <actor>Someone</actor>
  </cast>
  </movie>
    ...
</movies>

(person.tsv)
id  name    lastname    age
1   Janez   Novak   20
2   Ana Horvat  16
3   Eva Cerar   25

(tabela comment_connect_mysql na oddaljenem MySQL)
+-------------+------------+------+
| movie       | comment    | id   |
+-------------+------------+------+
| Star Wars   | odlicen    |    4 |
| Kekec       | odlicen    |    4 |
| 21          | negledljiv |    1 |
| Kekec       | napet      |    3 |
|   ..........................   |
+---------------------------------+
```

Slika 6.8: Struktura datotek movies.xml, person.tsv in tabele comment_connect_mysql na oddaljenem strežniku.

```
CREATE TABLE movies_connect (
language varchar (10) field_format='@lang',
title varchar (50) field_format='title',
year integer field_format='year',
rating integer field_format='rating',
actor varchar (50) field_format='cast/actor',
genre varchar (50) field_format='genre'
)ENGINE=CONNECT table_type=XML
file_name='/vagrant/maria/movies.xml'
tabname='movies'
option_list='rownode=movie,skipnull=1';
```

Slika 6.9: Koda za generiranje tabele iz datoteke movies.xml.

```
ResultSet rs = st.executeQuery("SELECT c.coment,
COUNT(*) FROM person_connect p, movies_connect m,
comments_connect c WHERE p.id=c.id AND
m.title=c.movie AND m.rating>7 AND p.age>20
GROUP BY c.comment");
```

Slika 6.10: Primer povezovanja tabel v Connectu.

6.8 Shranjevalni mehanizem Cassandra

Kvantitativna ocena

V tem testu smo primerjali, kolikšna je razlika v času, če izvajamo osnovne operacije CRUD na MariaDB tabelah in na MariaDB tabelah, ki so povezane s tabelami v sistemu Cassandra (tabela 6.4). Ker smo v testu vstavljali po eno vrstico naenkrat, MariaDB vrne slabe rezultate (moč je reševati z načini, opisanimi v [127]). V tem primeru je vstavljanje v tabele v sistemu Cassandra hitrejše, saj se ukazi pretvorijo iz MariaDB v ustrezne ukaze v sistemu Cassandra in se nato tam izvedejo.

Kvalitativna ocena

Shrambeni mehanizem Cassandra omogoča dostop in povezovanje tabel v sistemu Cassandra z MariaDB tabelami. V tem testu bomo pokazali, kako se poveže tabeli in kako sprememba v MariaDB tabeli povzroči spremembo v tabeli v sistemu Cassandra.

Na sistemu Cassandra izvedemo sledeče ukaze (slika 6.11). Na strani MariaDB moramo namestiti vtičnik in ustvariti tabelo, ki se bo povezovala s tabelo v sistemu Cassandra (slika 6.12). Sprememba v tabeli MariaDB povzroči spremembo na

operacija	MariaDB tabele	tabele v sistemu Cassandra	št. operacij
vstavljanje	47 s	36 s	50000
spreminjanje	okoli 0 s	61 s	50000
brisanje	okoli 0 s	58 s	100
poizvedovanje	20 s	43 s	100

Tabela 6.4: Primerjava operacij CRUD nad MariaDB tabelami in MariaDB tabelami, ki so povezane s tabelami sistema Cassandra.

```
CREATE KEYSPACE "cassandra" WITH REPLICATION={'class':'SimpleStrategy','replication_factor':1};

CREATE COLUMNFAMILY person(pid BIGINT PRIMARY KEY, name VARCHAR, LASTNAME VARCHAR)
with compact storage;

INSERT INTO cassandra.person (pid, name, lastname) VALUES (1, 'Janez', 'Novak');
INSERT INTO cassandra.person (pid, name, lastname) VALUES (2, 'Ana', 'Horvat');

cqlsh:cassandra> SELECT * FROM person ;

 pid | lastname | name
-----+----------+-------
   2 |   Horvat |   Ana
   1 |    Novak | Janez
```

Slika 6.11: Koda za ustvarjanje, vstavljanje in poizvedovanje v sistemu Cassandra.

```
install soname 'ha_cassandra.so';
set global cassandra_default_thrift_host='192.168.1.103';

CREATE TABLE person_cass1 (
pid BIGINT primary key,
name varchar(20),
lastname varchar(30)
)ENGINE=CASSANDRA thrift_host='192.168.1.103' keyspace='cassandra' column_family='person';

select * from person_cass1;
+-----+-------+----------+
| pid | name  | lastname |
+-----+-------+----------+
|   2 | Ana   | Horvat   |
|   1 | Janez | Novak    |
+-----+-------+----------+
```

Slika 6.12: Koda, ki jo moramo napisati na MariaDB strani, da se povežemo s sistemom Cassandra.

tabeli v sistemu Cassandra (slika 6.13). Iz dokumentacije smo razbrali, da je možno izvesti stik med tabelami [128] in izvajati osnovne operacije SELECT, INSERT, DELETE in UPDATE [42]. Prav tako vir [42] opisuje preslikovanje osnovnih operacij med sistemoma MariaDB in Cassandra.

```
MariaDB:
insert into person_cass1 VALUES (3, 'Miha', 'Cerar');

Cassandra:
cqlsh:cassandra> select * from person;
 pid | lastname | name
-----+----------+-------
   2 |   Horvat |   Ana
   3 |    Cerar |  Miha
   1 |    Novak | Janez
```

Slika 6.13: Vpisovanje v MariaDB tabelo povzroči vpis na tabelo v sistemu Cassandra.

6.9 Shranjevalni mehanizem TokuDB

Kvantitativna ocena

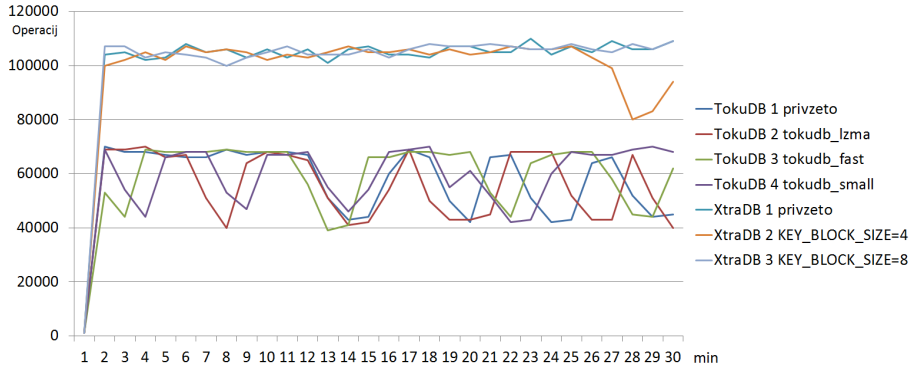
Ker smo teste opravljali na MariaDB različici 10.1, smo uporabili privzeti shranjevalni mehanizem XtraDB.

V prvem testu smo primerjali hitrost vstavljanja shranjevalnih mehanizmov TokuDB in XtraDB z različnimi nastavitvami kompresije. V drugem testu smo primerjali kompresijo med obema mehanizmoma. V tretjem testu smo primerjali čas, ki ga shranjevalna mehanizma potrebujeta, da dodata stolpec v tabeli z 10 000 000 zapisi.

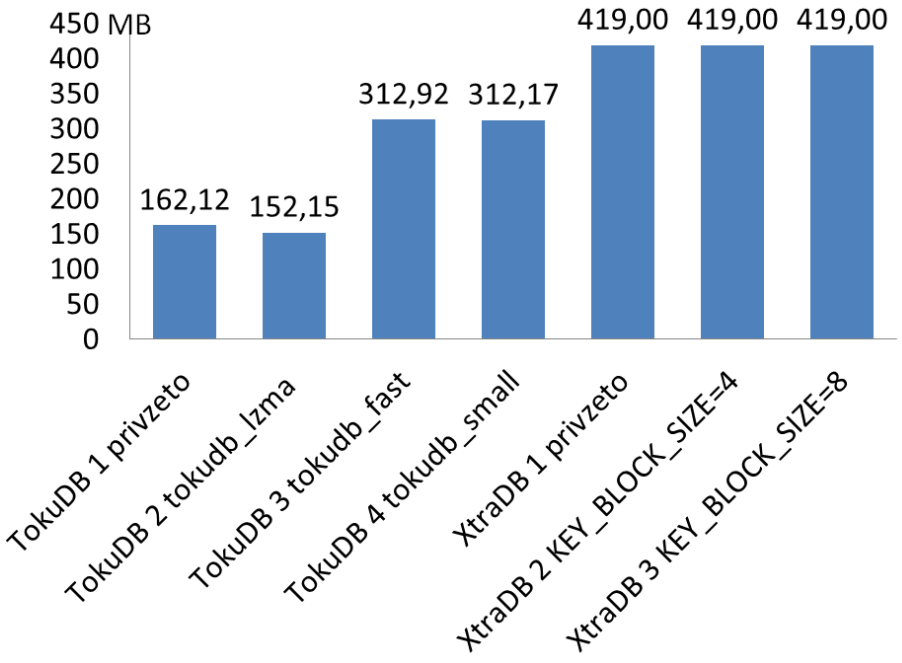
Prihajamo do ugotovitev, da TokuDB ne dosega rezultatov, objavljenih s strani podjetja Tokutek. XtraDB omogoča hitrejše operacije vstavljanja napram TokuDB. TokuDB po drugi strani omogoča veliko večjo kompresijo napram XtraDB, v najboljšem primeru je kompresija več kot dvakrat večja. Pri podjetju Tokutek utemeljujejo, da je ključ kompresiranje večjih blokov kot pri XtraDB. Izjemna razlika med XtraDB in TokuDB pa se pojavi pri dodajanju stolpca pri 10 milijonih zapisov. Podrobna razlaga je omenjena v 3.1.3.

dodajanje stolpca	čas	št. zapisov
HCAD	475 s	10000000
InnoDB	23456 s	10000000

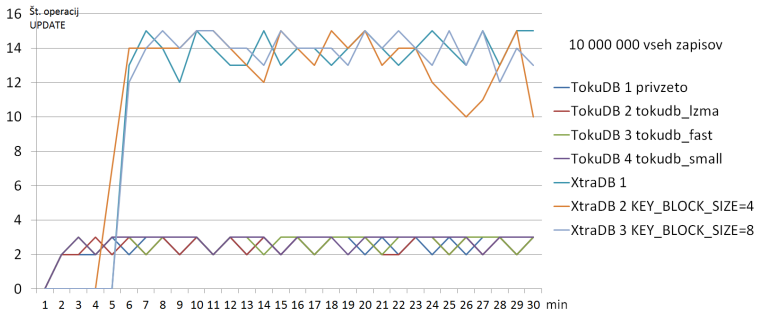
Tabela 6.5: Čas, potreben za dodajanje stolpcev.



Slika 6.14: Časovni pregled števila operacij vstavljanja na minuto med shranjevalnima mehanizmoma XtraDB in TokuDB.



Slika 6.15: Primerjava kompresije med XtraDB in TokuDB pri 10 milijonih zapisov.



Slika 6.16: Primerjava hitrosti operacij spreminjana podatkov (UPDATE) med XtraDB in TokuDB na množici 10 000 000 zapisov.

6.10 Shranjevalni mehanizem OQGraph

Kvalitativna ocena

V razdelku 3.1.4 smo prikazali izvedbo grafa z mehanizmom OQGraph. V tem delu bomo za namen primerjave prikazali, kako se isti problem rešuje s shrambo Neo4j.

Neo4j je linearno skalabilna (za operacije branja) [129] ACID shramba, ki shranjuje podatke v obliki grafov. Linearna sklabilnost je mogoča, ker vsaka instanca v gruči vsebuje isti graf (graf repliciran na vse instance v gruči), saj grafa ni mogoče fizično razdeliti na različne instance. V okviru projekta Rassilon razvijajo načine, kako bi graf porazdelili na več instanc [130]. Neo4j vsebuje vozlišča z lastnostmi, povezave in lastnosti povezav. Omogoča več načinov komunikacije s podatkovno bazo preko poizvedovalnega jezika Cypher, Jave, REST storitev in množice drugih jezikov, kot so PHP, Python, Ruby in drugi. V ukazni vrstici upravljamo z jezikom Cypher, ki je dobro podprt z dokumentacijo.

Neo4j je trenutno daleč najbolj uporabljen sistem za grafe [131], zato bomo prikazali njegovo razliko v primerjavi z OQGraph. V primerjavi z OQGraph za primer 3.6 za Neo4j potrebujemo akcije, kot je opisano na sliki 6.17.

Neo4j je bil razvit z namenom reševanja vprašanj, povezanih z grafi in z namenom obvladovanja hierarhij podatkov. To je njegova prednost pred OQGraph, ki je bil zgrajen nad obstoječo relacijsko shemo. OQGraph ne podpira kompleksnih poizvedb (v primerjavi z Neo4j) in ACID transakcij, kljub temu pa prinaša dodatno uporabno "NoSQL" funkcionalnost v MariaDB poleg relacijskih lastnosti.

```
CREATE (cristian:Actor {name: "Christian Bake", age:45})
CREATE (kateann:Actor {name: "Kate Ann-Moz", age: 38})
CREATE (rihard:Actor {name: "Rihard Din Anderson", age: 53})
...
CREATE (cristian)-[:KNOWS {years: 15}]->(kateann)
CREATE (kateann)-[:KNOWS]->(cristian)
CREATE (kateann)-[:KNOWS]->(scarl)
...
MATCH (kateann:Actor {name:"Kate Ann-Moz"}),
      (helen:Actor {name:"Helen Pagy"}),
      p=shortestPath((kateann)-[*]->(helen))
RETURN p
```

Slika 6.17: Primer kode v jeziku Cypher, ki poišče najkrajšo pot med igralcema "Kate Ann-Moz" in "Helen Pagy" za graf iz slike 3.6.

```
select GROUP_CONCAT(name ORDER BY seq)
AS "Veriga poznanstva" FROM actors JOIN
fix_graph ON (fix_graph.linkid=actors.id)
WHERE latch='dijkstra' AND origid=2
AND destid=8 \G
************* 1. row *******************
Veriga poznanstva: Kate Ann-Moz,Dani Craigl,
Orlando Bum,Joseb Gordom,Helen Pagy
```

Slika 6.18: Primer poizvedbe v OQGraph, ki ugotovi najkrajšo pot med vozliščema 2 (origid) in 8 (destid) za graf iz slike 3.6.

6.11 JSONB

Kvantitativna ocena

PostgreSQL je z novim podatkovnim tipom prodrl v področje, kjer je dosedaj prevladoval MongoDB. V tem testu smo primerjali hitrost pri poizvedovanju med PostgreSQL in MongoDB. Za dostop do SUPB smo uporabili Java knjižnice. V prvem koraku smo poizvedovali nad neindeksirano množico, v drugem koraku pa nad indeksirano. Poizvedbe smo ločili v dve množici: na tiste, ki so si po hitrosti podobne, in na tiste, ki imajo dolg izvajalni čas in kažejo izrazite slabosti sistema v primerjavi z drugim sistemom. Sledi primer zapisa v podatkovnih bazah.

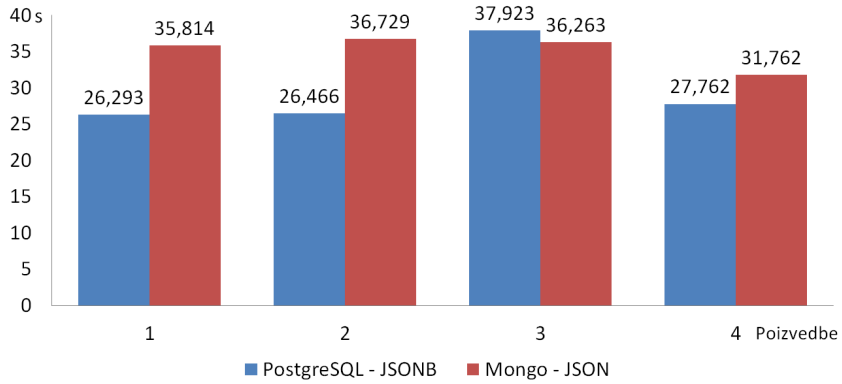
Pri testiranju se izkaže, da PostgreSQL hitreje odkrije podatke na neindeksirani množici (slika 6.20). Pri uporabi indeksov oba sistema izrazito izboljšata učinkovitost (slika 6.22). Bolje se izkaže MongoDB, ki izboljša rezultat in je hitrejši od PostgreSQL. Slabost PostgreSQL je uporaba ukaza DISTINCT, predvsem v kombinaciji z ukazom COUNT. MongoDB se slabo izkaže, ko je treba izpisati vse zapise. Drugo skupino testov sestavljajo poizvedbe, ki so se slabo izkazale na enem izmed sistemov (slika 6.23).

Kvalitativna ocena

Prednost PostgreSQL je, da uporablja SQL. Za delo z dokumenti JSON je uvedel nove operatorje, ki jih je možno uporabiti v stavkih SQL. Za dostop do elementov se uporabljata operatorja `->` in `->>`. Pri preverjanju vsebovanosti se uporablja operator `@>`. Moteča je nenehna uporaba pretvarjanja v podatkovne tipe, npr. `::int`. MongoDB po drugi strani omogoča različna tipa poizvedovanja: osnovno poizvedovanje z JavaScriptom in poizvedovanje Map-Reduce za obdelavo velike količine podatkov v uporabne agregirane rezultate. V magistrskem delu smo uporabili osnovno poizvedovanje z JavaScriptom. Pri izvajanju poizvedb moramo posredovati podatke v obliki JSON, ki lahko vsebujejo operatorje `$in, $or, $where, $gt` in ostale. Pomanjkljivost MongoDB je, da znotraj istega dokumenta ne moremo primerjati dveh podatkov med sabo. Za kaj takega moramo uporabiti agregacijske metode.

```
{
    "name" : "Janez"
    "lastname" : "Novak",
    "age" : 20,
    "commune" : "Ljubljana",
    "food" : [ "sir", "maslo" ],
    "earnings" : {
      "income" : 1500,
      "outcome" : 200
      }
}
```

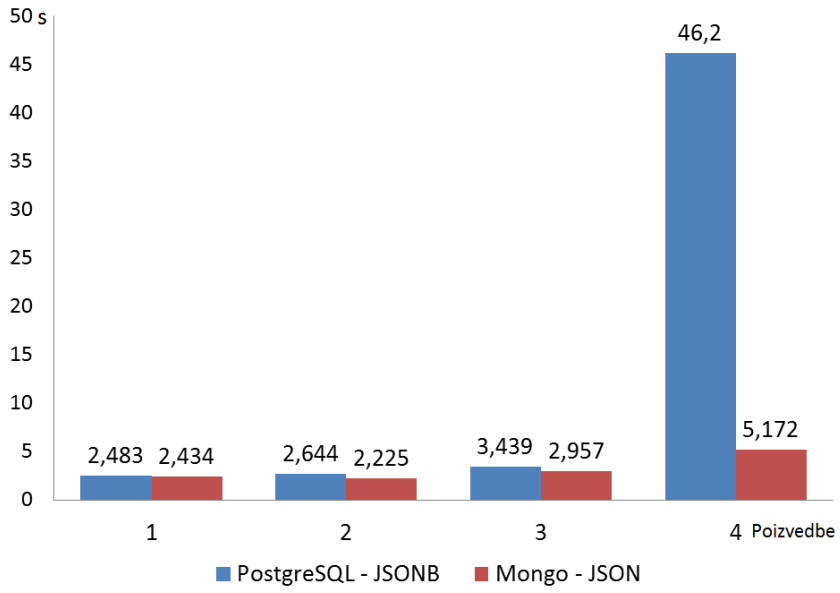
Slika 6.19: Format JSON, nad katerim smo poizvedovali.



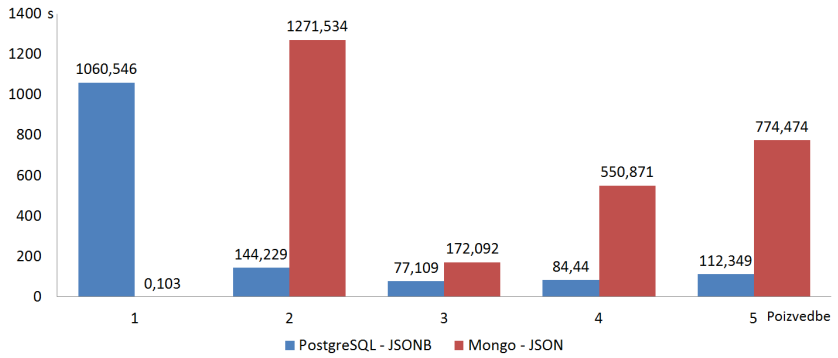
Slika 6.20: Primerjava hitrosti na neindeksirani množici (50 poizvedb). Poizvedbe so navedene na sliki 6.21.

```
1.
SELECT data->'name' FROM test WHERE data @> '{"commune" : "Ajdovščina" }'
db.test.find({ "commune" : "Ajdovščina"});

2.
SELECT data->'name'  FROM test WHERE data @> '{"commune" : "Ajdovščina" }'
and (data->>'age')::int > 50;
db.test.find({ "commune" : "Ajdovščina" , "age" : { "$gt" : 50}});

3.
SELECT data->'name'  FROM test WHERE data @> '{"commune" : "Ajdovščina",
"food" : ["maslo"] }' and (data->>'age')::int > 50;
db.test.find({ "commune" : "Ajdovščina" , "age" : { "$gt" : 50} , "food" :
{ "$in" : [ "maslo"]}});

4.
SELECT COUNT(*) FROM test where data @> '{"lastname" : "Novak"}' and
(data->'earnings'->>'outcome')::int > 2000;
db.test.find({ "earnings.outcome" : { "$gt" : 2000} , "lastname" : "Novak"});
```

Slika 6.21: Poizvedbe, ki se navezujejo na sliki 6.20 in 6.22.



Slika 6.22: Primerjava hitrosti na indeksirani množici (400 poizvedb). Poizvedbe so navedene na sliki 6.21.



Slika 6.23: Primerjava hitrosti težavnih poizvedb na indeksirani množici (50 poizvedb). Poizvedbe so navedene na sliki 6.24.

```
1.
SELECT distinct data->>'commune' from test;
db.test.distinct("commune");

2.
SELECT data->'name' FROM TEST;
db.test.find({ });

3.
SELECT data->'name'  FROM test where (data->'earnings'->>'income')::int =
 (data->'earnings'->>'outcome')::int;
{ "aggregate" : "test" , "pipeline" : [ { "$project" : { "cmp_value" : { "$cmp" : [
"$earnings.income" , "$earnings.outcome"]}}} , { "$match" : { "cmp_value" :
{ "$eq" : 0}}}]}

4.
SELECT data->'name'  FROM test where (data->>'age')::int > 50;
db.test.find({ "age" : { "$gt" : 50}});

5.
SELECT data->'name'  FROM test where (data->'earnings'->>'income')::int > 1500;
db.test.find({ "earnings.income" : { "$gt" : 1500}});
```

Slika 6.24: Problematične poizvedbe, ki se navezujejo na sliko 6.23.

Poglavje 7

Sklepne ugotovitve

Na področju podatkovnih baz se trenutno veliko razvija in raziskuje. Če upoštevamo evidenco s strani db-engines.com, obstaja vsaj 238 SUPB, ki so uvrščeni znotraj 13 kategorij (relacijski, iskalniki, NoSQL in drugi). V okviru magistrskega dela smo pregledali, katere značilnosti NoSQL vpeljujejo najvplivnejši odprtokodni sistemi PostgreSQL, MariaDB in MySQL, pri čemer smo dodali še arhitekturo za skaliranje. Pregledali smo tudi prenašanje konceptov v sisteme NoSQL in VoltDB kot predstavnika NewSQL sistemov.

7.1 Obrazložitev hipotez

V poglavju 2 smo predpostavili tri hipoteze o trenutnem stanju, katere lahko sedaj ovrednotimo.

(a) **Konvergenca se zaustavlja, saj se zdi, da se prenašajo le značilnosti, ki so zanimive; čaka se na naslednje zanimive koncepte.**

Po raziskovanju smo mnenja, da hipoteza drži. Razviti so dodatki v obliki vtičnikov in shranjevalnih mehanizmov, a ne moremo trditi, da gre za zbliževanje obeh polov, saj večjih sprememb v arhitekturi z izjemo FoundationDB ni.

(b) **Konvergenca poteka bolj v smeri podpore nerelacijskih konceptov v relacijskih sistemih, kjer se nerelacijski koncepti dodajajo predvsem kot dodatki (vtičniki).**

Hipoteza ne drži v celoti. Zdi se, da ponudniki NoSQL ne čutijo potrebe prodreti v smer relacijskih sistemov, saj je ta del zrel in zelo močan, poleg tega pa je implementacija standardiziranega poizvedovalnega jezika in ACID transakcij zelo težavna, čeprav mogoča (FoundationDB). Čeprav so HandlerSocket, Memcached API in shranjevalni mehanizmi OQGraph, TokuDB, Connect in Cassandra implementirani kot vtičniki, imamo tudi primera, kjer so koncepti implementirani kot podatkovni tipi, npr. dinamični stolpci, JSONB, hstore.

(c) **Sistemi NewSQL predstavljajo sintezo obojih sistemov, a sistemi NewSQL niso rezultat konvergence relacijskih in nerelacijskih sistemov, ki zmanjša razlike med poloma, temveč produkt, ki vsebuje omejen del relacijskih in del nerelacijskih lastnosti.**

Iz pregleda VoltDB in FoundationDB sklepamo, da sistemi NewSQL vsebujejo sintezo sistemov, a v omejeni obliki. Za FoundationDB lahko trdimo, da stopa v smer konvergence, saj so povezali arhitekturo, značilno za NoSQL (ključ-vrednost), z relacijskim modelom na višji plasti in omogočili uporabo SQL nad dejansko shrambo tipa ključ-vrednost. Za VoltDB ne moremo trditi, da gre v smeri konvergence. Res je, da v relacijske sisteme vpeljuje nerelacijske koncepte, kot sta obdelava podatkov v pomnilniku in uporaba particioniranja, toda novi koncepti povzročajo tudi večji razkorak z relacijskimi koncepti (omejena uporaba SQL).

Konvergenca sistemov je mogoča, saj na to nakazujeta FoundationDB in PostgreSQL z JSONB, a zdi se, da izrazitih sprememb razen dodajanja določenih funkcij ne bo. V FoundationDB so na tem področju naredili največ, saj so dokazali, da je porazdeljeni sistem NoSQL moč nadgraditi z relacijskim modelom in zagotoviti ACID transakcije z ohranjanjem dobrih rezultatov. Pri pregledovanju novic večjih ponudnikov ni zaslediti, da bi se kakšno večje podjetje odločilo izrazito spremeniti svoj sistem. Na področju nerelacijskih sistemov manjka standardni poizvedovalni jezik, kot je SQL za relacijske sisteme. Trenutno nismo zasledili veliko pobud po standardiziranem povpraševalnem jeziku za nerelacijske sisteme, a določeni sistemi so razvili jezike, ki se zgledujejo po SQL sintaksi, npr. CQL, GQL in N1QL.

7.2 Povzetek ugotovitev o prenesenih konceptih

Memcached API ni dosegel naših pričakovanj. Ni vsebovan v vsaki izdaji MySQL, njegova hitrost izvajanja operacij se močno razlikuje glede na uporabljene gonilnike in treba je določiti tabelo, kamor se bodo shranjevali podatki. Kot pozitivno lahko navedemo, da je moč poizvedovati po tabeli s SQL. **HandlerSocket** je dosegel svoj namen, saj ponuja hitre operacije, ni pa intuitiven v primerjavi s SQL in zavrže s SQL povezano validacijo podatkov. **Dinamični stolpci** so sicer dodali podporo nestrukturiranim podatkom, a njihova izvedba še ni izpopolnjena. Manjkajo nekatere lastnosti JSON in indeksiranje brez virtualnih stolpcev, poleg tega pa niso hitri pri operacijah vstavljanja in poizvedovanja. Shranjevalna mehanizma **Connect** in **Cassandra** delujeta kot uporabni dopolnitvi sistema MariaDB. **Shranjevalni mehanizem TokuDB** prinaša mešane občutke. Po eni strani omogoča večjo kompresijo podatkov in hitrejše dodajanje stolpcev ter indeksov, a po drugi strani ne dosega hitrosti operacij shranjevalnega mehanizma InnoDB (XtraDB). **OQGraph** je primerna rešitev za relacijske sisteme, ko želimo reševati običajna vprašanja, povezana z grafi (iskanje najkrajše poti), a za kompleksnejša ni primeren. PostgreSQL je z dodanim podatkovnim tipom **JSONB** po našem mnenju naredil velik korak, saj so operacije hitre ter dodana sintaksa v SQL razumljiva in hitro naučljiva. Pregledali smo, kako **MySQL Fabric** rešuje problem skaliranja in prišli do ugotovitev, da ima ranljivosti. Zanimiv pristop je **Dynomite**, ki pa se še razvija. Po našem mnenju je največji preskok na področju konvergence naredilo podjetje **FoundationDB**, saj je omogočilo relacijski model in SQL-92 nad shrambo tipa ključ-vrednost. Poleg tega so prikazali, da je moč zagotoviti ACID transakcije in pri tem še vedno ohranjati učinkovitost operacij. **VoltDB** opredeljujemo kot zanimiv izdelek, ki želi nadomestiti zasnovo relacijskih sistemov iz 70-ih let prejšnjega stoletja. Prinaša všečne koncepte (obdelave podatkov v pomnilniku, uporaba nedeljene arhitekture in druge), a najbolj moteča je po našem mnenju uporaba standarda SQL-99 s posebnostmi.

7.3 Kritika in možne izboljšave

Kot prednost magistrskega dela bi izpostavili, da so koncepti opisani tudi s primeri programske kode, ki koncepte bolje razjasni. Kot slabosti magistrskega dela bi izpostavili, da rezultati primerjalnih testov temeljijo na enostavnih testih, ki pa niso standardizirani. Testi so bili izvedeni v virtualnem okolju na prenosniku, kar morda ne odraža najbolj točnega rezultata. Poleg tega predpostavke glede NewSQL temeljijo predvsem na ugotovitvah iz sistemov VoltDB in FoundationDB. Za bolj objektivno stanje bi bilo treba pregledati še ostale sisteme, npr. Clustrix in NuoDB.

7.4 Možnosti za nadaljnje raziskovanje

Na tem področju je še marsikaj, česar v tem magistrskem delu nismo obdelali. Zanimivo področje, ki se trenutno raziskuje, je SQL++ [132], katerega namen je razširiti SQL za delo s polstrukturiranimi podatki. Prav tako v tem delu nismo raziskali t.i. shramb z več modeli (multi-model databases). Gre za shrambe, ki niso le relacijskega tipa ali tipa ključ-vrednost, ampak podpirajo še grafe, dokumente in ostale tipe. Čeprav se je YCBS uveljavil kot "de facto" orodje za izvajanje testov, pa bi bilo zanimivo pogledati, ali se kaj raziskuje na področju standardizacije primerjalnih testov za NoSQL. Nadalje je tu rešitev Google F1 [133], ki naj bi bila skalabilna porazdeljena relacijska podatkovna baza, katere izvedba plasti SQL je zelo podobna izvedbi plasti SQL pri FoundationDB.

Literatura

[1] Webscale PostgreSQL - JSONB and Horizontal Scaling Strategies. Dostopno na: http://www.slideshare.net/jkatz05/webscale-postgresql-jsonb. Datum: 2014-10-27.

[2] Cory Nance, Travis Losser, Reenu Iype, and Gary Harmon. NoSQL vs RDBMS - Why There is Room for Both. In *Proceedings of the Southern Association for Information Systems Conference (SAIS)*, 2013.

[3] Wikipedia - Scalability. Dostopno na: `http://en.wikipedia.org/wiki/Scalability`. Datum: 2015-02-14.

[4] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[5] DB-Engines: DBMS popularity broken down by database model. Dostopno na: `http://db-engines.com/en/ranking_categories`. Datum: 2014-11-14.

[6] Oracle Database 12c Overview. Dostopno na: `http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html`. Datum: 2015-02-14.

[7] IBM DB2 database software. Dostopno na: `http://www-01.ibm.com/software/data/db2/`. Datum: 2015-02-14.

[8] Microsoft Cloud Platform - SQL Server 2014. Dostopno na: `http://www.microsoft.com/en-us/server-cloud/products/sql-server/`. Datum: 2015-02-14.

[9] MySQL. Dostopno na: http://www.mysql.com/. Datum: 2015-02-14.

[10] MariaDB - The Best of Both Worlds. Dostopno na: https://mariadb.com/blog/best-both-worlds. Datum: 2014-10-13.

[11] PostgreSQL. Dostopno na: http://www.postgresql.org/. Datum: 2015-02-14.

[12] Andrew Oppel. *Databases Demystified*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2004.

[13] A. B. M. Moniruzzaman and Syed Akhter Hossain. NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *CoRR*, abs/1307.0191, 2013.

[14] Shashank Tiwari. *Professional NoSQL*. Wrox programmer to programmer. John Wiley, Hoboken, N.J. Wiley Chichester, 2011. Index.

[15] DB-Engines Ranking of Key-value Stores. Dostopno na: http://db-engines.com/en/ranking/key-value+store. Datum: 2015-02-14.

[16] DB-Engines Ranking of Document Stores. Dostopno na: http://db-engines.com/en/ranking/document+store. Datum: 2015-02-14.

[17] DB-Engines Ranking of Wide Column Stores. Dostopno na: http://db-engines.com/en/ranking/wide+column+store. Datum: 2015-02-14.

[18] DB-Engines Ranking of Graph DBMS. Dostopno na: http://db-engines.com/en/ranking/graph+dbms. Datum: 2015-02-14.

[19] DB-Engines Ranking. Dostopno na: http://db-engines.com/en/ranking. Datum: 2015-03-07.

[20] Amazon DynamoDB. Dostopno na: http://aws.amazon.com/dynamodb/. Datum: 2015-03-07.

[21] Cassandra - Welcome to Apache Cassandra. Dostopno na: http://cassandra.apache.org/. Datum: 2014-09-19.

[22] Explanation of BASE terminology. Dostopno na: `http://stackoverflow.com/questions/3342497/explanation-of-base-terminology`. Datum: 2014-10-01.

[23] Introduction to NOSQL databases. Dostopno na: `http://stelmach.biz/blog/2012/post2.html`. Datum: 2012-05-07.

[24] Microsoft researchers: NoSQL needs standardization. Dostopno na: `http://www.computerworld.com/article/2507783/database-administration/microsoft-researchers--nosql-needs-standardization.html`. Datum: 2014-10-01.

[25] OLTP vs OLAP. Dostopno na: `http://www.slideshare.net/fmhyudin/oltp-vs-olap-23317601`. Datum: 2014-10-03.

[26] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, Feb 2010.

[27] Are there any NoSQL standards emerging. Dostopno na: http://stackoverflow.com/questions/3439878/are-there-any-nosql-standards-emerging. Datum: 2014-10-01.

[28] Redis - Transactions. Dostopno na: http://redis.io/topics/transactions. Datum: 2015-02-15.

[29] UnQL Query Language Unveiled by Couchbase and SQLite. Dostopno na: `http://www.couchbase.com/press-releases/unql-query-language`. Datum: 2015-02-15.

[30] ArangoDB - Is UNQL Dead? Dostopno na: `https://www.arangodb.com/2012/04/07/is_unql_dead`. Datum: 2015-02-15.

[31] Introducing Dynomite - Making Non-Distributed Databases, Distributed. Dostopno na: http://techblog.netflix.com/2014/11/introducing-dynomite.html. Datum: 2014-11-10.

[32] FoundationDB - SQL Layer Concepts, Known Limitations. Dostopno na: https://foundationdb.com/layers/sql/documentation/Concepts/known.limitations.html. Datum: 2014-12-19.

[33] Google Cloud Platform - GQL Reference. Dostopno na: `https://cloud.google.com/appengine/docs/python/datastore/gqlreference`. Datum: 2015-02-15.

[34] NewSQL: what's this? Dostopno na: http://labs.sogeti.com/newsql-whats/. Datum: 2014-11-09.

[35] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[36] Percona XtraDB Software - Enhanced InnoDB for MySQL. Dostopno na: `http://www.percona.com/software/percona-server/percona-xtradb`. Datum: 2015-02-19.

[37] MariaDB - Introduction to Connect engine. Dostopno na: `https://mariadb.com/kb/en/mariadb/documentation/storage-engines/connect/introduction-to-the-connect-engine/`. Datum: 2014-09-16.

[38] MariaDB - Inward and Outward Tables. Dostopno na: https://mariadb.com/kb/en/mariadb/documentation/storage-engines/connect/connect-table-types/inward-and-outward-tables/. Datum: 2014-10-18.

[39] Using CONNECT - General Information. Dostopno na: https://mariadb.com/kb/en/mariadb/documentation/storage-engines/connect/using-connect/using-connect-general-information/. Datum: 2014-09-16.

[40] MariaDB 10 CONNECT Engine - A Better Way to Access External Data. Dostopno na: https://mariadb.com/blog/mariadb-10-connect-engine-better-way-access-external-data. Datum: 2014-09-16.

[41] MariaDB CONNECT Engine - Fosdem 2014. Dostopno na: `https://archive.fosdem.org/2014/schedule/event/mariadb_connect_engine/attachments/slides/420/export/events/attachments/`

mariadb_connect_engine/slides/420/FOSDEM2014_MariaDB_CONNECT_Storage_Engine.pdf. Datum: 2014-09-18.

[42] Cassandra Storage Engine Overview. Dostopno na: https://mariadb.com/kb/en/mariadb/cassandra-storage-engine-overview/. Datum: 2014-09-18.

[43] MariaDB Cassandra Interoperability. Dostopno na: http://www.slideshare.net/SergeyPetrunya/mysqlconf2013-mariadbcassandrainteroperability. Datum: 2014-09-19.

[44] TokuDB for MySQL. Dostopno na: http://www.tokutek.com/products/tokudb-for-mysql/. Datum: 2014-09-22.

[45] May 2nd Webinar: Introduction to TokuDB v7 Community. Dostopno na: http://www.tokutek.com/2013/04/may-2nd-webinar-introduction-to-tokudb-v7-community-enterprise-editions/. Datum: 2014-09-24.

[46] TokuMX Fractal Tree(R) indexes, what are they? Dostopno na: http://www.tokutek.com/2013/07/tokumx-fractal-treer-indexes-what-are-they/. Datum: 2014-09-22.

[47] Fractal Tree Indexes : From Theory to Practice. Dostopno na: http://www.slideshare.net/tmcallaghan/20131112-plukfractaltreestheorytopractice. Datum: 2015-03-10.

[48] TokuDB V6.0: Even better compression. Dostopno na: http://www.tokutek.com/2012/04/tokudb-v6-0-even-better-compression/. Datum: 2014-09-23.

[49] TokuDB Documentation. Dostopno na: http://docs.tokutek.com/tokudb/. Datum: 2015-03-08.

[50] Hot Column Addition and Deletion Part 1. Dostopno na: http://www.tokutek.com/2011/03/hot-column-addition-and-deletion-part-i-performance/. Datum: 2014-09-24.

[51] Hot Column Addition and Deletion Part 2. Dostopno na: http://www.tokutek.com/2011/04/hot-column-addition-and-deletion-part-ii-how-it-works/. Datum: 2014-09-24.

[52] Introducing Multiple Clustering Indexes. Dostopno na: http://www.tokutek.com/2009/05/introducing_multiple_clustering_indexes/. Datum: 2014-09-24.

[53] TokuDB vs InnoDB HDD. Dostopno na: http://www.tokutek.com/resources/benchmark-results/benchmarks-vs-innodb-hdds/#slavelag. Datum: 2014-09-24.

[54] TokuDB - Hot Indexing Part 1: New Feature. Dostopno na: http://www.tokutek.com/2011/04/hot-indexing-part-i-new-feature/. Datum: 2014-09-24.

[55] What are the pros and cons of these MySQL engines for OLTP - XtraDB, PBXT, or TokuDB? Dostopno na: http://serverfault.com/questions/130521/what-are-the-pros-cons-of-these-mysql-engines-for-oltp-xtradb-pbxt-or-tok. Datum: 2015-02-19.

[56] Detailed review of Tokutek storage engine. Dostopno na: http://www.percona.com/blog/2009/04/28/detailed-review-of-tokutek-storage-engine/. Datum: 2015-02-19.

[57] Neo4j. Dostopno na: http://neo4j.com/. Datum: 2015-02-19.

[58] MariaDB - OQGRAPH Overview. Dostopno na: https://mariadb.com/kb/en/mariadb/documentation/storage-engines/oqgraph-storage-engine/oqgraph-overview/. Datum: 2014-10-05.

[59] Neo4j Manual - Find people based on similar favorites. Dostopno na: http://neo4j.com/docs/stable/cypher-cookbook-similar-favorites.html. Datum: 2014-11-28.

[60] MySQL 5.6 tackles NoSQL competitors. Dostopno na: http://www.networkworld.com/article/2163277/applications/mysql-5-6-tackles-nosql-competitors.html. Datum: 2014-09-11.

[61] Memcached. Dostopno na: `http://memcached.org/`. Datum: 2014-10-21.

[62] Guide to NoSQL with MySQL. Dostopno na: `http://www.slideshare.net/Akasam42/guide-to-nosql-with-mysql`. Datum: 2014-09-11.

[63] Internals of the InnoDB memcached Plugin. Dostopno na: `http://dev.mysql.com/doc/refman/5.6/en/innodb-memcached-internals.html`. Datum: 2014-10-21.

[64] 4.18.5.4 Controlling Transactional Behavior of the InnoDB memcached Plugin. Dostopno na: `http://dev.mysql.com/doc/refman/5.6/en/innodb-memcached-txn.html`. Datum: 2015-02-15.

[65] 14.18.5.6 Performing DML and DDL Statements on the Underlying InnoDB Table. Dostopno na: `http://dev.mysql.com/doc/refman/5.6/en/innodb-memcached-ddl.html`. Datum: 2015-02-15.

[66] Benefits of the InnoDB / memcached Combination. Dostopno na: `http://dev.mysql.com/doc/refman/5.6/en/innodb-memcached-benefits.html`. Datum: 2014-10-21.

[67] Getting Started with the MariaDB HandlerSocket Plugin. Dostopno na: https://mariadb.com/blog/getting-started-mariadb-handlersocket-plugin. Datum: 2014-09-08.

[68] Using MySQL as a NoSQL - A story for exceeding 750,000 qps on a commodity server. Dostopno na: `http://yoshinorimatsunobu.blogspot.com/2010/10/using-mysql-as-nosql-story-for.html`. Datum: 2014-09-08.

[69] MySQL Limitations Part 4: One thread per connection. Dostopno na: http://www.percona.com/blog/2010/10/27/mysql-limitations-part-4-one-thread-per-connection/. Datum: 2014-10-08.

[70] MariaDB - HandlerSocket Client Libraries. Dostopno na: `https://mariadb.com/kb/en/mariadb/handlersocket-client-libraries/`. Datum: 2015-02-15.

[71] MySQL+HandlerSocket=NoSQL (Percona London 2012). Dostopno na: `https://www.youtube.com/watch?v=Q1HRY8y6O9Q`. Datum: 2014-09-09.

[72] MariaDB Dynamic Columns. Dostopno na: https://mariadb.com/kb/en/mariadb/mariadb-documentation/nosql/dynamic-columns/. Datum: 2014-07-17.

[73] Using JSON with MariaDB and MySQL. Dostopno na: http://www.slideshare.net/blueskarlsson/using-json-with-mariadb-and-mysql. Datum: 2014-09-15.

[74] Hstore. Dostopno na: http://postgresguide.com/sexy/hstore.html. Datum: 2014-09-07.

[75] MariaDB - Virtual (Computed) Columns. Dostopno na: https://mariadb.com/kb/en/mariadb/virtual-computed-columns/. Datum: 2015-02-16.

[76] PostgreSQL 9.2.10 Documentation - XML Type! Dostopno na: http://www.postgresql.org/docs/9.2/static/datatype-xml.html. Datum: 2015-02-16.

[77] XML, HSTORE, JSON, JSONB - OH MY! Dostopno na: http://www.pgcon.org/2014/schedule/attachments/313_xml-hstore-json.pdf. Datum: 2015-02-16.

[78] Christophe Pettus. PostgreSQL as a Schemaless Database. Fosdem 2013, Brussels, Belgium, 2013.

[79] PostgreSQL Documentation - JSON Types. Dostopno na: http://www.postgresql.org/docs/9.4/static/datatype-json.html. Datum: 2014-09-07.

[80] PostgreSQL Documentation - Functions JSON. Dostopno na: http://www.postgresql.org/docs/9.4/static/functions-json.html. Datum: 2014-10-28.

[81] Json - what's new in 9.4, and what's left to do. Dostopno na: https://www.youtube.com/watch?v=MmzbnMqBMI0. Datum: 2014-09-07.

[82] MySQL Cluster Core Concepts. Dostopno na: http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-basics.html. Datum: 2015-02-17.

[83] MySQL - Fabric FAQ. Dostopno na: `http://www.mysql.com/products/enterprise/fabric/faq.html`. Datum: 2014-12-03.

[84] MySQL Cluster NDB 7.3 and MySQL Cluster NDB 7.4. Dostopno na: `http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster.html`. Datum: 2014-12-03.

[85] MySQL - Chapter 2. The NDB API. Dostopno na: `http://dev.mysql.com/doc/ndbapi/en/ndbapi.html`. Datum: 2015-02-17.

[86] Known Limitations of MySQL Cluster. Dostopno na: `http://dev.mysql.com/doc/refman/5.6/en/mysql-cluster-limitations.html`. Datum: 2014-12-03.

[87] MySQL - Sharding Scenario. Dostopno na: `http://dev.mysql.com/doc/mysql-utilities/1.4/en/fabric-quick-start-sharding-scenario.html`. Datum: 2014-11-10.

[88] Sharding and Scale-out using MySQL Fabric. Dostopno na: http://www.slideshare.net/mkindahl/sharding-and-scaleout-using-mysql-fabric. Datum: 2014-11-10.

[89] Oracle Database In-Memory. Dostopno na: http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html. Datum: 2015-02-19.

[90] JSON in Oracle Database. Dostopno na: `https://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6275`. Datum: 2015-02-19.

[91] DB2 NoSQL support. Dostopno na: `http://www-01.ibm.com/software/data/db2/linux-unix-windows/nosql-support.html`. Datum: 2015-02-19.

[92] Scaling Out SQL Server. Dostopno na: `https://msdn.microsoft.com/en-us/library/aa479364.aspx`. Datum: 2015-02-19.

[93] Achieving Scalability and Availability with Peer-to-Peer Transactional Replication. Dostopno na: `https://technet.microsoft.com/en-us/library/cc966404.aspx`. Datum: 2015-02-19.

[94] How to Effectively Map SQL Data to a NoSQL Store. Dostopno na: http://www.infoq.com/articles/map-sql-nosql. Datum: 2014-11-11.

[95] The Key-Value Store makes your architecture flexible and easy to operate. Dostopno na: https://foundationdb.com/key-value-store/architecture. Datum: 2014-11-11.

[96] FoundationDB - SQL Layer Features. Dostopno na: https://foundationdb.com/layers/sql/documentation/Concepts/features.html. Datum: 2015-02-17.

[97] FoundationDB - Table Groups. Dostopno na: https://foundationdb.com/layers/sql/documentation/Concepts/table.groups.html. Datum: 2014-11-11.

[98] FoundationDB - Known Limitations. Dostopno na: https://foundationdb.com/key-value-store/documentation/known-limitations.html. Datum: 2014-12-19.

[99] FoundationDB Blog - Databases at 14.4MHz. Dostopno na: http://blog.foundationdb.com/databases-at-14.4mhz. Datum: 2014-12-19.

[100] Clustrix. Dostopno na: http://www.clustrix.com/. Datum: 2015-02-18.

[101] NuoDB. Dostopno na: http://www.nuodb.com/. Datum: 2015-02-18.

[102] VoltDB technical overview. Dostopno na: http://voltdb.com/downloads/datasheets_collateral/technical_overview.pdf. Datum: 2014-10-08.

[103] MemSQL. Dostopno na: http://www.memsql.com/. Datum: 2015-02-18.

[104] VoltDB tutorials. Dostopno na: http://voltdb.com/resources/volt-university/tutorials/. Datum: 2014-10-07.

[105] VoltDB Documentation - Chapter 10. Availability. Dostopno na: http://docs.voltdb.com/UsingVoltDB/ChapKSafety.php. Datum: 2015-3-08.

[106] VoltDB Decapitates Six SQL Urban Myths And Delivers Internet Scale OLTP In The Process. Dostopno na:

http://highscalability.com/blog/2010/6/28/voltdb-decapitates-six-sql-urban-myths-and-delivers-internet.html. Datum: 2014-10-29.

[107] FoundationDB - ACID transactions are incredibly helpful. Dostopno na: `https://foundationdb.com/acid-claims`. Datum: 2015-03-07.

[108] VoltDB Query Optimization. Dostopno na: `http://voltdb.com/resources/lessons/17-voltdb-query-optimization`. Datum: 2014-10-29.

[109] Performance Guide - Reading the Execution Plan and Optimizing Your SQL Statements. Dostopno na: `http://docs.voltdb.com/PerfGuide/ExecPlansRead.php`. Datum: 2014-10-29.

[110] Using VoltDB - CREATE TABLE. Dostopno na: `https://voltdb.com/docs/UsingVoltDB/ddlref_createtable.php`. Datum: 2014-10-08.

[111] Using VoltDB - SELECT. Dostopno na: `http://docs.voltdb.com/UsingVoltDB/sqlref_select.php`. Datum: 2014-10-29.

[112] Database benchmark wars: What you need to know. Dostopno na: http://www.techrepublic.com/article/database-benchmark-wars-what-you-need-to-know/. Datum: 2014-11-17.

[113] Overview of the TPC-C Benchmark The Order-Entry Benchmark. Dostopno na: `http://www.tpc.org/tpcc/detail.asp`. Datum: 2014-11-24.

[114] BenchmarkSQL. Dostopno na: `http://sourceforge.net/projects/benchmarksql/`. Datum: 2015-02-19.

[115] MySQL Benchmark Tool - DBT2 Benchmark Tool. Dostopno na: `http://dev.mysql.com/downloads/benchmarks.html`. Datum: 2015-02-19.

[116] HammerDB. Dostopno na: `http://www.hammerdb.com/`. Datum: 2015-02-19.

[117] TPCC-UVa: A free, open-source implementation of the TPC-C Benchmark. Dostopno na: `http://www.infor.uva.es/~diego/tpcc-uva.php`. Datum: 2015-02-19.

[118] Yanpei Chen, Francois Raab, and Randy H. Katz. From TPC-C to Big Data Benchmarks: A Functional Workload Model. Technical Report UCB/EECS-2012-174, EECS Department, University of California, Berkeley, Jul 2012.

[119] How fast is Redis? Dostopno na: `http://redis.io/topics/benchmarks`. Datum: 2014-12-10.

[120] MongoDB - mongoperf. Dostopno na: `http://docs.mongodb.org/manual/reference/program/mongoperf/`. Datum: 2014-12-10.

[121] Benchmarking Top NoSQL Databases. Dostopno na: `https://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf`. Datum: 2014-12-10.

[122] Solving the Fast Data Problem for the Cloud. Dostopno na: `http://voltdb.com/sites/default/files/voltdb\_softlayer\_datasheet.pdf`. Datum: 2014-12-10.

[123] Apache Cassandra NoSQL Performance Benchmarks. Dostopno na: `http://planetcassandra.org/nosql-performance-benchmarks/`. Datum: 2014-12-10.

[124] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[125] J. Zhan, R. Han, and C. Weng. *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 4th and 5th Workshops, BPOE 2014, Salt Lake City, USA, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised Selected Papers*. Lecture Notes in Computer Science / Information Systems and Applications, incl. Internet/Web, and HCI. Springer International Publishing, 2015.

[126] Vagrant. Dostopno na: `https://www.vagrantup.com/`. Datum: 2015-02-18.

[127] MariaDB - How to Quickly Insert Data Into MariaDB. Dostopno na: https://mariadb.com/kb/en/mariadb/how-to-quickly-insert-data-into-mariadb/. Datum: 2015-02-20.

[128] MariaDB - Handling Joins With Cassandra. Dostopno na: `https://mariadb.com/kb/en/mariadb/handling-joins-with-cassandra/`. Datum: 2015-02-19.

[129] Understanding Neo4j Scalability. Dostopno na: `http://info.neo4j.com/rs/neotechnology/images/Understanding\%20Neo4j\%20Scalability(2).pdf`. Datum: 2015-02-19.

[130] Neo4j Blog 2013: What's Coming Next in Neo4j! Dostopno na: `http://neo4j.com/blog/2013-whats-coming-next-in-neo4j/`, note = Datum: 2015-02-19.

[131] DB-Engines Ranking of Graph DBMS. Dostopno na: `http://db-engines.com/en/ranking/graph+dbms`. Datum: 2014-11-29.

[132] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases.

[133] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, et al. F1: the fault-tolerant distributed RDBMS supporting google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM, 2012.